VAX Pascal

digital

Reference Supplement for VMS Systems

Reference Supplement for VMS Systems

Order Number AA-PASYA-TE

VAX Pascal Reference Supplement for VMS Systems

Order Number: AA–PASYA–TE

December 1989

This manual provides information on the programming environment beyond the scope of the VAX Pascal language syntax and semantics. It contains reference information on VMS operating-system features, VAX architecture information, and environment-specific affects of VAX Pascal language features.

Revision/Update Information:	This is a new manual. However, most of the information in this manual was previously located in the *VAX PASCAL User Manual* (Order Number AI–H485E–TE).
Operating System and Version:	VMS Version 5.2 or higher.
Software Version:	VAX Pascal Version 4.0.

digital equipment corporation
maynard, massachusetts

December 1989

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

CDA	MASSBUS	VAX RMS
DDIF	PrintServer 40	VAXstation
DEC	Q-bus	VMS
DECnet	ReGIS	VT
DECUS	ULTRIX	XUI
DECwindows	UNIBUS	
Digital	VAX	
LN03	VAXcluster	digital™

The following are third-party trademarks:

PostScript is a registered trademark of Adobe Systems, Inc.

ZK5218

Contents

Tables

Preface

This document is the complete description of the VAX Pascal programming language. It contains information on the VAX Pascal language syntax and semantics, on VAX Pascal adherence to various Pascal standards, and on the VAX Pascal extensions to those standards.

You can use the reference information in this manual to write programs or modules that are only used on the VMS operating system on a VAX machine (VAX Pascal programs that are not intended to be portable). If you need to write VAX Pascal programs or modules that must be compiled by another compiler, see the *VAX Pascal Reference Manual* for a checklist of language extensions not included in the Pascal standard. The *VAX Pascal Reference Manual* also provides information on the Pascal standard.

Intended Audience

This manual is intended for experienced applications programmers with a basic understanding of the Pascal language. Some familiarity with your operating system is helpful. This is not a tutorial manual.

Document Structure

This manual consists of the following chapters and appendixes:

- Chapter 1 provides information on compiling programs, linking programs, running programs, and using text and object-module libraries.
- Chapter 2 provides information on data storage formats and value ranges for objects of given VAX Pascal data types.
- Chapter 3 provides information on the VAX Procedure Calling Standard as applied to VAX Pascal programs.

- Chapter 4 provides information on VAX Pascal system definitions files, and declaring and calling system routines.
- Chapter 5 provides information on the relationship between VAX Pascal input and output, and the underlying VAX Record Management Services (RMS) constructs.
- Chapter 6 provides information on the implementation of VAX Pascal predeclared routines and identifiers.
- Chapter 7 provides information on error processing and the writing of condition handlers.
- Chapter 8 provides information on tools that you may want to use to develop VAX Pascal programs.
- Appendix A provides a checklist of VAX Pascal implementation features that have environment-specific affects.
- Appendix B provides descriptions for diagnostic messages that can be generated by the VAX Pascal compiler and Run-Time System.
- Appendix C provides a list of possible error values returned by the STATUS and STATUSV functions.
- Appendix D provides a list of entry points to utilities in the VAX Run-Time Library that can be called as external routines by a VAX Pascal program.

Associated Documents

The following documents may also be useful when programming in VAX Pascal:

- *VAX Pascal Reference Manual*—Provides information on the syntax and semantics of the VAX Pascal programming language.
- *VAX Pascal User Manual*—Provides information about programming tasks, about using VAX Pascal features in conjunction with one another, and about increasing the efficiency of program execution.
- *VAX Pascal Installation Guide*—Provides information on how to install VAX Pascal on your operating system.
- VMS operating system manuals—Provide full information about the system. The *VMS Master Index* briefly describes all VMS system documentation, defines the intended audience for each manual, and provides a synopsis of each manual's contents.

Conventions

Table 1 presents the conventions used in this manual.

Table 1: Conventions Used in This Manual

Convention	Meaning
{ }	Large braces enclose lists from which you must choose one item. For example: { expression } { statement }
. . .	A horizontal ellipsis means that the item preceding the ellipsis can be repeated. For example: digit . . .
{ }, . . .	Braces followed by a comma and a horizontal ellipsis mean that you can repeat the enclosed item one or more times, separating two or more items with commas. For example: {label}, . . .
{ }; . . .	Braces followed by a semicolon and a horizontal ellipsis mean that you can repeat the enclosed item one or more times, separating two or more items with semicolons. For example: REPEAT {statement}; . . . UNTIL expression
. . .	A vertical ellipsis in a figure or example means that not all of the statements are shown.
[ ]	Square brackets mean that the statement syntax requires the square bracket characters. This notation is used with arrays, sets, and attribute lists. For example: ARRAY[index1]

(continued on next page)

Table 1 (Cont.): Conventions Used in This Manual

Convention	Meaning
[[ ]]	Double brackets enclose items that are optional. For example: EOLN [[(file-variable)]]
PROGRAM WRITELN	Uppercase letters and special symbols in syntax descriptions indicate VAX Pascal reserved words and predeclared identifiers. For example: BEGIN END
temp : INTEGER; PRED(n)	Lowercase letters represent elements that you must replace according to the description in the text.
module	A term that appears in bold is defined in the glossary in the *VAX Pascal User Manual*.
$ PASCAL $_File:	The hardcopy version of this manual has interactive examples that show user input in red letters and system responses or prompts in black letters. The online version differentiates user input from system responses by displaying the user input in bold text.

In this manual, complex examples and syntax diagrams have been divided into several lines to make them easy to read. Pascal does not require that you format your programs in any particular way. You should not regard the formats used in this manual as mandatory.

Chapter 1

Program Compilation and Execution

After you use a text editor to write code to a file, you need to use Digital Command Language (DCL) commands to compile modules and programs, and to link and run programs. This chapter provides information on the following topics:

- The PASCAL command (Section 1.1)
- The LINK command (Section 1.2)
- The RUN command (Section 1.3)
- Error-message format (Section 1.4)

For More Information:

For information on DCL syntax, see HELP or the *VMS DCL Concepts Manual*.

1.1 PASCAL Command

The PASCAL command invokes the VAX Pascal compiler, which verifies program source statements, issues error messages, and generates and groups machine language instructions into an object module for the linker.

PASCAL [[{/command-qualifier} . . .]] {file-spec[[{/file-qualifier} . . .]]} { + , } . . .

/qualifier [[= { file-spec, identifier, (identifier, . . .), n }]]

/command-qualifier
The name of a qualifier that indicates special processing to be performed by the compiler on all files listed.

file-spec
The name of one of the following:

- The input source file that contains the program or module to be compiled. If you separate multiple source file specifications with commas, the programs are compiled separately. If you separate the file specifications with plus signs, the files are concatenated and compiled as one program.

  The default file type for an input file is either .PAS (source file) or .TLB (text-library module).
- The output file, used only with the /ANALYSIS_DATA, /ENVIRONMENT, /LIST, /OBJECT, or /DIAGNOSTICS qualifiers.

/file-qualifier
The name of a qualifier that indicates special processing to be performed by the compiler on the files to which the qualifier is attached.

identifier
The name of one or more options that modify only the /CHECK, /DEBUG, /DESIGN, /OPTIMIZE, /SHOW, /STANDARD, /TERMINAL, and /USAGE qualifiers.

n
The integer constant, used only with the /ERROR_LIMIT qualifier, that indicates the maximum number of errors to be detected before compilation ceases.

The default for compiler output files (object modules) is the .OBJ file type.

Consider the following examples:

```
$ PASCAL/LIST [DIR]M
```

The source file M.PAS in directory [DIR] is compiled, producing an object file named M.OBJ and a listing file named M.LIS. The compiler places the object and listing files in the default directory.

```
$ PASCAL/LIST A, B, C
```

Source files A.PAS, B.PAS, and C.PAS are compiled as separate files, producing object files named A.OBJ, B.OBJ, and C.OBJ, and listing files named A.LIS, B.LIS, and C.LIS.

```
$ PASCAL X + Y + Z
```

Source files X.PAS, Y.PAS, and Z.PAS are concatenated and compiled as one file, producing an object file named X.OBJ. In batch mode, this command also produces the listing file X.LIS.

```
$ PASCAL A, B, C+D/LIST, F
```

When a qualifier follows the file specification, it applies only to the file immediately preceding it. Files concatenated with plus signs are considered one file. This command line produces four object files, A.OBJ, B.OBJ, C.OBJ, and F.OBJ, and one listing file, D.LIS.

```
$ PASCAL A + CIRC/NOOBJECT + X
```

The command shown above completely suppresses the object file; that is, source files A.PAS, CIRC.PAS, and X.PAS are concatenated and compiled, but no object file is produced.

1.1.1 PASCAL Qualifiers

This section describes the command and file qualifiers that you can use when compiling code.

You can also include attributes in your source code to alter the default actions of the compiler. (Portability concerns and application requirements determine whether you should use qualifiers or attributes.) For instance, if you want to compile the program using G_floating format for DOUBLE variables, you can apply the /G_FLOATING qualifier to the PASCAL command, or you can include the following attribute in your source program:

```
[G_FLOATING]    PROGRAM  Record_Keeping;
```

/[NO]ANALYSIS_DATA
/NOANALYSIS_DATA (default)
Creates a file containing source code analysis information. If you omit the file specification, the analysis file defaults to the name of your source file with a .ANA file type. The source code analysis file is reserved for use with VMS Layered Products such as, but not limited to, the VAX Source Code Analyzer (SCA).

/[NO]CHECK
/CHECK=(BOUNDS,DECLARATIONS) (default)

Directs the compiler to generate code to perform run-time checks. A single identifier or a list of identifiers enclosed in parentheses may follow /CHECK; these identifiers are the names of options that tell the compiler which aspects of the compilation unit to check.

The system issues an error message and normally terminates execution if any of the conditions in the options list occur. Table 1–1 lists the available checking options, their corresponding actions, and their negations.

Table 1–1: /CHECK Qualifier Options

Option	Action	Negation
ALL	Generates checking code for all options.	NONE
BOUNDS	Verifies that an index expression is within the bounds of an array's index type, that character-string sizes are compatible with the operations being performed, and that schemata are compatible.	NOBOUNDS
CASE_SELECTORS	Verifies that the value of a case selector is contained in the corresponding case-label list.	NOCASE_SELECTORS
DECLARATIONS	Verifies that schema definitions yield valid types and that uses of GOTO from one block to an enclosing block are correct.	NODECLARATIONS
OVERFLOW	Verifies that the result of an integer computation does not exceed the machine representation.	NOOVERFLOW

(continued on next page)

Table 1–1 (Cont.): /CHECK Qualifier Options

Option	Action	Negation
POINTERS	Verifies that the value of a pointer variable is not NIL.	NOPOINTERS
SUBRANGE	Verifies that values assigned to variables of subrange types are within the subrange; verifies that a set expression is assignment compatible with a set variable; verifies that MOD operates on positive numbers.	NOSUBRANGE

The BOUNDS and DECLARATIONS options are the only checking options enabled by default. The /CHECK qualifier without options is equivalent to /CHECK=ALL. The negation /NOCHECK is equivalent to /CHECK=NONE.

The CHECK attribute in the source program or module overrides the /CHECK qualifier on the command line.

/[NO]CROSS_REFERENCE
/NOCROSS_REFERENCE (default)
Produces a cross-reference section within the listing file. The compiler ignores this qualifier if you do not also specify /LIST on the same command line.

/[NO]DEBUG
/DEBUG=TRACEBACK (default)
Specifies that the compiler is to generate information for use by the VMS Debugger and the run-time error traceback mechanism. A single identifier or a list of identifiers enclosed in parentheses may follow /DEBUG; these identifiers are the names of options that inform the compiler which type of information it should generate.

Table 1–2 lists the available options, their corresponding actions, and their negations.

Table 1–2: /DEBUG Qualifier Options

Option	Action	Negation
ALL	Specifies that the compiler should include symbol and traceback information in the object module.	NONE
SYMBOLS	Specifies that the compiler should include in the object module symbol definitions for all identifiers in the compilation.	NOSYMBOLS
TRACEBACK	Specifies that the compiler should include in the object module traceback information permitting virtual addresses to be translated into source program routine names and compiler-generated line numbers.	NOTRACEBACK

When you specify SYMBOLS without TRACEBACK, the table of compiler-generated line numbers is omitted from the debugger symbol table.

You should consider using /NOOPTIMIZE when you are using /DEBUG. Allowing optimizations to occur may make debugging difficult and may obscure some sections of the compilation unit that you would like to debug.

The /DEBUG qualifier without options is equivalent to /DEBUG=ALL. The negation /NODEBUG is equivalent to /DEBUG=NONE.

/[NO]DESIGN
/NODESIGN (default)

Directs the compiler to accept design phase placeholders and comments as valid program elements within a VAX Pascal program. Placeholders are produced by you or the VAX Language-Sensitive Editor (LSE); design comments are intended for use with the VAX Source Code Analyzer (SCA). To use /DESIGN, you must be running LSE Version 3.0 or higher and SCA Version 2.0 or higher. Table 1–3 lists the available options, their corresponding actions, and their negations.

Table 1–3: /DESIGN Qualifier Options

Option	Action	Negation
PLACEHOLDERS	Directs the compiler to accept placeholders as valid program elements.	NOPLACEHOLDERS
COMMENTS	Directs the compiler to recognize design comments.	NOCOMMENTS

The /DESIGN qualifier without options is equivalent to /DESIGN=(PLACEHOLDERS, COMMENTS).

/[NO]DIAGNOSTICS
/NODIAGNOSTICS (default)
Creates a file containing compiler messages and diagnostic information. If you omit the file specification, the diagnostics file defaults to the name of your source file with a .DIA file type. The diagnostics file is reserved for use with VMS Layered Products such as, but not limited to, the VAX Language-Sensitive Editor (LSE).

/[NO]ENVIRONMENT
determined by attributes (default)
Produces an environment file in which declarations and definitions made at the outermost level of a compilation unit are saved. The default file name is the same as the source file name. The default file type is .PEN, an abbreviation for "Pascal Environment." You can provide a different name for the environment file by including a VMS file specification after the /ENVIRONMENT qualifier, as in /ENVIRONMENT=MASTER.PEN.

The /ENVIRONMENT qualifier on the command line overrides the ENVIRONMENT attribute in the source program or module. By default, the attributes of the source program or module determine whether an environment file is created; however, if the /ENVIRONMENT qualifier is specified at compile time, an environment file will always be created.

/[NO]ERROR_LIMIT
/ERROR_LIMIT=30 (default)
Terminates compilation after the occurrence of a specified number of error messages, excluding warning-level and information-level errors. If you specify /NOERROR_LIMIT, compilation continues until 500 errors have been detected.

/[NO]G_FLOATING
/NOG_FLOATING (default)
Directs the compiler to use the G_floating representation and instructions for values of type DOUBLE. The negation of this qualifier specifies the use of the D_floating representation and instructions.

If the use of the /G_FLOATING qualifier conflicts with a double-precision attribute specified in the source program or module, a warning occurs. Routines and compilation units between which double-precision quantities are passed should not mix the D_floating and G_floating data types. Not all processors support the G_floating data type.

file-spec/LIBRARY
none (default)
Specifies that a file is a text library file. The text library file specification is required. The text library files in a list of source files must be concatenated by plus signs. The default file type is .TLB.

/[NO]LIST
/NOLIST (interactive default)
/LIST=input_file_name.LIS (batch default)
Produces a source listing file with a file type of .LIS. See the /SHOW qualifier for more information on controlling the contents of the source listing file.

/[NO]MACHINE_CODE
/NOMACHINE_CODE (default)
Produces a machine code section within the listing file. If the compiler detects errors in the source code, the compiler does not generate this section. The compiler ignores this qualifier if you do not also specify /LIST on the same command line.

/[NO]OBJECT
/OBJECT=input_file_name.OBJ (default)
Specifies the name of the object file. If the compiler detects errors in the source code, the compiler writes no representation of object code to the listing file.

/[NO]OLD_VERSION
/NOOLD_VERSION (default)
Directs the compiler to resolve differences between VAX Pascal Version 1.0 and subsequent versions by using the Version 1.0 definition of the language.

/[NO]OPTIMIZE
/OPTIMIZE (default)
Directs the compiler to optimize the code for the program or module being compiled so that the compiler generates more efficient code. A single identifier or a list of identifiers enclosed in parentheses may follow /OPTIMIZE; these identifiers are the names of options that tell the compiler which aspects of the compilation unit to optimize.

Table 1–4 lists the available options, their corresponding actions, and their negations.

Table 1–4: /OPTIMIZE Qualifier Options

Option	Action	Negation
ALL	Enables all optimization components.	NONE
INLINE	Enables inline expansion of user-defined routines.	NOINLINE

The /OPTIMIZE qualifier without options is equivalent to /OPTIMIZE=ALL. The negation /NOOPTIMIZE is equivalent to /OPTIMIZE=NONE.

Some VAX Pascal programs that run under VAX Pascal Version 1.0, however, may require the use of the /NOOPTIMIZE qualifier in order to execute the same way with subsequent versions.

The OPTIMIZE and NOOPTIMIZE attributes in the source program or module override the /OPTIMIZE and /NOOPTIMIZE qualifiers on the command line.

The /NOOPTIMIZE qualifier guarantees left-to-right evaluation order with full evaluation of both operands of the AND and OR Boolean operators to aid in diagnosing all potential programming errors. If you wish to have short circuit evaluation even with the /NOOPTIMIZE qualifier, use the AND_THEN and OR_ELSE Boolean operators.

/[NO]SHOW
/SHOW=(DICTIONARY,HEADER,INCLUDE,SOURCE,STATISTICS, TABLE_OF_CONTENTS) (default)
Specifies a list of items to be included in the listing file. A single identifier or a list of identifiers enclosed in parentheses may follow /SHOW; these identifiers are the names of options that inform the compiler which type of information it should generate.

Table 1–5 lists the available options, their corresponding actions, and their negations.

Table 1–5: /SHOW Qualifier Options

Option	Action	Negation
ALL	Enables listing of all options.	NONE
DICTIONARY	Enables listing of %DICTIONARY records.	NODICTIONARY
HEADER	Enables page headers.	NOHEADER
INCLUDE	Enables listing of %INCLUDE files.	NOINCLUDE
INLINE	Enables listing of inline summary.	NOINLINE
SOURCE	Enables listing of VAX Pascal source code.	NOSOURCE
STATISTICS	Enables listing of compilation statistics.	NOSTATISTICS
TABLE_OF_ CONTENTS	Enables listing of a table of contents only if the %TITLE or %SUBTITLE directive was specified in the source code.	NOTABLE_OF_ CONTENTS

The inline summary, enabled by the /SHOW=INLINE qualifier, shows only the names of routines that were expanded inline in the compilation. If you want to know why routines were not expanded inline, you must specify an additional qualifier, either /OPTIMIZE=INLINE or /OPTIMIZE=ALL. Although /OPTIMIZE defaults to /OPTIMIZE=ALL, you must explicitly specify the ALL option to generate these reasons.

The compiler ignores the /SHOW qualifier if you do not also specify the /LIST qualifier on the same command line. The negation /NOSHOW is equivalent to /SHOW=NONE; /SHOW is equivalent to /SHOW=ALL.

/[NO]STANDARD
/NOSTANDARD (default)
Causes the compiler to generate messages wherever the compilation unit uses VAX Pascal language extensions, which are nonstandard Pascal features. Within the VAX Pascal documentation set, these standards are collectively referred to as the "Pascal standard."

Table 1–6 lists the available options, their corresponding actions, and their negations.

Table 1–6: /STANDARD Qualifier Options

Option	Action	Negation
NONE	Disables standards checking.	N.A.
ANSI	Uses the rules of the ANSI standard.	N.A.
ISO	Uses the rules of the ISO standard.	N.A.
EXTENDED	Uses the rules of the Extended standard.	N.A.
VALIDATION	Performs validation for the given standard.	NOVALIDATION

The /STANDARD qualifier allows you to use only two options. The first option selects the standard to be used (ANSI, ISO or EXTENDED). The second option determines whether the strict validation rules are to be enforced ([NO]VALIDATION). /STANDARD=(ANSI, ISO, VALIDATION) is not allowed because both ANSI and ISO are specified.

By default, these information-level messages are written to the error file SYS$ERROR. Note that using the VALIDATION option changes all nonstandard information-level messages to error-level messages.

The /STANDARD qualifier without options is equivalent to /STANDARD=(ANSI, NOVALIDATION). /STANDARD=VALIDATION is equivalent to /STANDARD=(ANSI, VALIDATION). The negation /NOSTANDARD is equivalent to /STANDARD=NONE.

/[NO]TERMINAL
/NOTERMINAL (default)

Specifies a list of items to be displayed on the terminal. A single identifier or a list of identifiers enclosed in parentheses may follow the /TERMINAL qualifier; these identifiers are options that inform the compiler which type of information to display.

Table 1–7 lists the available options, their corresponding actions, and their negations.

Table 1–7: /TERMINAL Qualifier Options

Option	Action	Negation
ALL	Displays all options.	NONE
FILE_NAME	Displays file names on Pascal command line as they are being processed.	NOFILE_NAME
ROUTINE_NAME	Displays routine names as code is generated.	NOROUTINE_NAME
STATISTICS	Displays compiler statistics.	NOSTATISTICS

The /TERMINAL qualifier without options is equivalent to /TERMINAL=ALL. The negation /NOTERMINAL is equivalent to /TERMINAL=NONE.

/[NO]USAGE
/USAGE=UNINITIALIZED (default)
Directs the compiler to perform compile-time checks indicated by the chosen options. A single identifier or a list of identifiers enclosed in parentheses may follow /USAGE; these identifiers are options that tell the compiler which checks to perform.

Table 1–8 lists the available options, their corresponding actions, and their negations.

Table 1–8: /USAGE Qualifier Options

Option	Action	Negation
ALL	Enables checking of all options.	NONE
UNCERTAIN	Checks for variables that may be uninitialized depending on program flow.	NOUNCERTAIN
UNINITIALIZED	Checks for variables that are known to be uninitialized.	NOUNINITIALIZED
UNUSED	Checks for variables that are declared but never referenced.	NOUNUSED

The following types of variables are not checked for uninitialization:

- Variables that have a file component
- Predeclared INPUT or OUTPUT identifiers

- Variables that have global, external, or inherited visibility
- Variables declared with the AT attribute
- Variables declared with the COMMON attribute
- Variables declared with the READONLY attribute
- Variables declared with the VOLATILE attribute
- Variables used as parameters
- Variables used as function identifiers

The /USAGE qualifier without options is equivalent to /USAGE=ALL. The negation /NOUSAGE is equivalent to /USAGE=NONE.

/[NO]WARNINGS
/WARNINGS (default)
Directs the compiler to generate diagnostic messages in response to warning-level or informational-level errors.

By default, these messages are written to the error file SYS$ERROR. A warning or informational diagnostic message indicates that the compiler has detected acceptable but unorthodox syntax or has performed some corrective action; in either case, unexpected results may occur.

Note that informational messages generated when you specify the /STANDARD qualifier do not appear if /NOWARNINGS is enabled.

For More Information:

- On debugging (Section 8.1)
- On text libraries (Section 1.1.9)
- On LSE and SCA information (Section 8.3)
- On error messages (Section 1.4)
- On the contents of a compiler listing (Section 1.1.2)
- On Pascal standards and on changes since VAX Pascal Version 1.0 (*VAX Pascal Reference Manual*)
- On using environment files (*VAX Pascal User Manual*)
- On the AND_THEN and OR_ELSE Boolean operators (*VAX Pascal Reference Manual*)

1.1.2 Contents of the Compilation Listing File

You control the contents of a compilation listing by appending qualifiers to the PASCAL command. Table 1–9 lists the six parts of a complete compilation listing and the qualifiers that cause them to be generated.

Table 1–9: Compilation Listing Contents and Qualifiers

Section	Generated With
Table of contents	/LIST /SHOW=TABLE_OF_CONTENTS
Source code	/LIST
Cross-reference	/LIST /CROSS_REFERENCE
Machine code	/LIST /MACHINE_CODE
Inline summary	/LIST /SHOW=INLINE
Compilation statistics	/LIST /SHOW=STATISTICS

A compilation listing file usually contains source code because the /SHOW=SOURCE qualifier is enabled by default. The /LIST qualifier does not initiate the printing of the listing file. To obtain a line printer copy of your listing file, use the PRINT command.

You can control the number of lines that appear on a listing page by defining the SYS$LP_LINES logical name before invoking the compiler. For example:

```
$ DEFINE SYS$LP_LINES 100
$ PASCAL/LIST [DIR]M
```

This set of commands creates a printed page size of 94 lines (the compiler subtracts six lines for margins).

The following sections describe the contents of each part of the listing file.

For More Information:

- On the PASCAL command qualifiers (Section 1.1.1)
- On the SYS$LP_LINES logical (*VMS Run-Time Library Routines Volume*)

1.1.3 Table of Contents

The table of contents lists the line number, listing page number, and source file page number of each section of the source code. These sections are delineated by %TITLE or %SUBTITLE directives that indicate the name by which the section is known; for example, "Main Program Body." The compiler ignores the /SHOW=TABLE_OF_CONTENTS qualifier if the compilation unit does not contain a %TITLE or %SUBTITLE directive.

1.1.4 Source Code

The source code part of a listing file includes source code line numbers (LINE column); a notation identifying %INCLUDE directive code, %DICTIONARY directive code, and comments (IDC column); a procedure nesting level (PL column); a statement nesting level (SL column); source code; and diagnostic messages. The following example shows some lines of a procedure accessed with an %INCLUDE directive:

```
(LINE)  (IDC)  (PL)  (SL)
00021     I     1     0     PROCEDURE PRINT (Arr : Arrtype);
00022     I     1     0        VAR I := INTEGER;
00023     I     1     1        BEGIN
```

1.1.5 Cross-Reference Section

The cross-reference part of a listing file contains a list of all identifiers and labels used within the source code. This list includes the name of the identifier or label, the program element it represents, the source code line numbers where it appears, and, where applicable, the attributes, declaring block, and function result type associated with it. The following example shows an entry for a procedure named Print:

```
PRINT     PROC     [PSECT ($CODE), UNBOUND]    {IN PROGRAM EXAMPLE}
                   21        41
```

1.1.6 Machine Code Section

The machine code part of a listing file contains a representation of the object code generated by the compiler. Information is organized by program section and, within each program section, by executable block.

For each program section, the compiler generates the program section name and properties. For each executable block, the compiler generates the information identified in Example 1–1. Note that the listing format is similar to, but not exactly like, macro; that is, if the section is edited to remove the hexidecimal notation on the left side, it will not assemble using the VAX MACRO assembler.

Example 1–1: Machine Code Section of a Compilation Listing

```
                                    (1)
  20  65  68   54      00000 C.AAA .ASCII  \The \
  65  20  68   74      00004 C.AAB .ASCII  \th element is \
                   01 00012       NOP
                   01 00013       NOP

                                    (2)          (3)
                 003C 00000 PRINT .WORD   ^M<R2,R3,R4,R5>

       (4)             (5)     (6)    (7)       (8)              (9)
5E                1C  C2 00002        SUBL2   #28,SP
         F8       AD  D4 00005        CLRL    -8(FP)
6D 00000000G      EF  9E 00008        MOVAB   PAS$HANDLER,(FP)
BC                14  28 0000F        MOVC3   #20,@4(R12),ARR
5C                01  DO 00015  1$:   MOVL    #1,R12          ; 0024
52                5C  DO 00018  2$:   MOVL    R12,I
   FFFFFFFCB      EF  9F 0001B        PUSHAB  C.AAA
                                        .
                                        .
                                        .
          (10)                    (11)
Routine Size: 100 bytes,  Routine Base: $CODE + 00098
```

(1) Names for literal constants, in the form C.AAA.

(2) Entry point (.ENTRY for global blocks, otherwise the program or routine name and .WORD).

(3) Save mask definition.

(4) Hexadecimal representation of the code, in right-to-left order. Operands marked with the letter G refer to symbols stored in a program section other than the current one or found in external routines.

(5) Current location counter. The counter for routines is expressed as an offset from the beginning of the routine; the counter for data is expressed as an offset from the beginning of the program section. To compute the absolute offset of a routine from the beginning of the program section, add the location counter to the routine base (see (11)).

(6) Computer-generated labels, used as the targets of compiler-generated instructions.

❼ Symbolic opcode.

❽ Symbolic operands (if needed).

❾ Comment section noting the source line number if the corresponding instruction is the first one generated for that source line.

❿ Routine size in bytes.

⓫ Routine base, expressed as an offset from the start of the program section.

1.1.7 Inline Summary

The inline summary part of a listing file contains a summary indicating which routine calls of user-defined routines were or were not expanded inline. This summary includes the name of the calling routine, program, or module; the call and line number of the call; and a notation indicating whether expansion occurred.

1.1.8 Compilation Statistics

The compilation statistics part of a listing file contains the following categories of summary information:

- *Psect Summary*, listing the psect name, number of bytes, and attributes of all program sections created during compilation.
- *Environment Statistics*, listing the names of all environment files inherited by the compilation and symbol information. This information includes the total number of symbols in the environment file, the number of symbols actually used by the compilation, and the percentage of used symbols versus defined symbols.

  Note that the VAX Pascal compiler defines "symbol" in terms of internal representation. This definition may not reflect the complexity of the environment source; that is, the number of symbols shown loaded may not reflect the number of symbols in your program.
- *Command Qualifiers and Options List*, containing the exact command line passed by DCL to the VAX Pascal compiler and the qualifier options in effect during compilation.
- *Compiler Internal Timing Statistics*, noting the number of page faults and amount of elapsed time and CPU time required for each phase of the compilation.

- *Compilation Statistics*, listing the total number of messages generated at each level—informational, warning, error, and fatal; the time and speed of compilation; and the number of page faults that occurred. The last line is a message indicating that the compilation of the source code is complete.

1.1.9 Text Libraries

A text library contains modules of source text that you can incorporate in a program by using the %INCLUDE directive. This directive indicates the module and, optionally, the text library in which the module can be found. Text library names can be specified in the following ways:

- In the %INCLUDE directive
- On the PASCAL command line
- In a DCL DEFINE default library command

1.1.9.1 Using the %INCLUDE Directive for Text Libraries

The %INCLUDE directive has the following form:

```
%INCLUDE '[[file-spec]] (module-name) [[/[[NO]]LIST ]] '
```

file-spec
The name of the text library containing a module to be included in the compilation.

module-name
The name of a text module, located in a text library, that is to be included in the source file. The name of the module must be enclosed in parentheses. The module names can include any printable character except a space, horizontal tab, comma, or exclamation point. The maximum length of the module name is determined when the text library is created. Module names are also case insensitive.

/[NO]LIST
Indicates that the included module should be printed in the listing of the program if a listing is being generated. If not specified, the default is determined by the [NO]INCLUDE option on the /SHOW qualifier. The INCLUDE option enables the listing of %INCLUDE files and is enabled by default.

For example, the following %INCLUDE directive specifies both the text library DATAB.TLB and the module External_Declarations:

```
%INCLUDE 'DATAB.TLB (External_Declarations)'
```

If the text library is not specified in the %INCLUDE directive, its name must appear on the PASCAL command line or it must be specified by a DCL DEFINE command.

For More Information:

- On /LIST and /SHOW qualifiers (Section 1.1.1)
- On module name restrictions (*VMS Librarian Utility Manual*)
- On default libraries (Section 1.1.9.3)

1.1.9.2 Specifying Text Libraries on the PASCAL Command Line

The /LIBRARY qualifier identifies text libraries specified on the PASCAL command line. When you compile a source file that includes a module from a text library, concatenate the name of the text library to the name of the source file and append the /LIBRARY qualifier. You specify concatenation with a plus sign. For example:

```
$ PASCAL APPLIC+DATAB/LIBRARY
```

This command instructs the compiler to search the DATAB text library each time it encounters an %INCLUDE directive within the APPLIC source file.

If more than one library is specified, the compiler searches the libraries in the order they appear on the command line. For example:

```
$ PASCAL APPLIC+DATAB/LIBRARY+DATAC/LIBRARY+DATAD/LIBRARY
```

If you request multiple compilations, the /LIBRARY qualifier must appear after each compilation in which it is needed. For example:

```
$ PASCAL METRIC+DATAB/LIBRARY, APPLIC+DATAB/LIBRARY
```

If you are concatenating source files, the /LIBRARY qualifier can appear only after the last source file. For example:

```
$ PASCAL METRIC.PAS+APPLIC.PAS+DATAB/LIBRARY
```

Any Pascal output qualifiers that appear after the /LIBRARY qualifier, such as /OBJECT or /LISTING, apply to the last source file name that you specified. For example, the following PASCAL command creates APPLIC.OBJ:

```
$ PASCAL METRIC+APPLIC+DATAB/LIBRARY/OBJECT
```

For More Information:

For information on the PASCAL command and qualifiers, see Section 1.1.

1.1.9.3 Defining Default Libraries

You can define one of your private text libraries as a default text library for the Pascal compiler to search. The VAX Pascal compiler searches the default library after it searches libraries specified in the PASCAL command.

To establish a default library, define the logical name PASCAL$LIBRARY, as in the following example of the DEFINE command:

```
$ DEFINE PASCAL$LIBRARY DISK$:[LIB]DATAB
```

While this assignment is in effect, the compiler automatically searches the library DISK$:[LIB]DATAB.TLB for any included modules that it cannot locate in libraries explicitly specified on the PASCAL command.

The VAX Pascal compiler uses PASCAL$LIBRARY as the file name for the default text library; the location and search order of the logical name tables are controlled by RMS.

If PASCAL$LIBRARY is defined as a search list, the compiler opens the first item specified in the list. If the include module is not found there, the search is terminated, and an error message is issued.

For More Information:

For information on the DCL DEFINE command, see the *VMS DCL Dictionary*.

1.2 LINK Command

The LINK command invokes the VMS Linker, which combines object modules into one executable image, which can then be executed by the VMS operating system.

LINK [[{/command-qualifier} . . .]] {file-spec[[{/file-qualifier} . . .]]} , . . .

/qualifier [[= { file-spec / library-module / (library-module, . . .) }]]

/command-qualifier
The name of a qualifier that indicates special processing to be performed by the linker on all files listed.

file-spec
The name of one of the following:

- The input file (which can be the name of an object module library) that contains the object code to be linked.
- The options file, used only with the /OPTIONS qualifier.
- The output file, used only with the /EXECUTABLE and /MAP qualifiers.

/file-qualifier
The name of a qualifier (the /INCLUDE, /LIBRARY, or /OPTIONS qualifier) that indicates special processing to be performed by the linker on the files to which the qualifier is attached.

library-module
The name of one or more object modules or shareable image libraries that you can only specify using the /INCLUDE or /LIBRARY qualifiers.

A source program or module cannot run on the system until it is linked. If you are using .PEN (Pascal Environment) files that include variables, procedures, or functions, make sure you link the object file into the .EXE file. When you execute the LINK command, the VMS Linker performs the following functions:

- Resolves local and global symbolic references in the object code
- Assigns values to the global symbolic references
- Signals an error message for any unresolved symbolic reference

The linker uses the name of the input file that you specified first on the command line for the name of the output file. The default for linker output files (executable images) is the .EXE file type.

Consider the following examples:

```
$ LINK DANCE.OBJ, CHACHA.OBJ, SWING.OBJ
```

This command links the object files DANCE.OBJ, CHACHA.OBJ, and SWING.OBJ to produce one executable image called DANCE.EXE.

```
$ LINK/EXECUTABLE=TEST CIRCLE
```

This command links CIRCLE.OBJ and then causes the executable image generated by the linker to be named TEST.EXE.

```
$ LINK SCHEDULE,COURSES/INCLUDE=(HISTORY,ALGEBRA,PHILOSOPHY)
```

This example shows the use of the /INCLUDE qualifier with a library named COURSES. The linker extracts the modules HISTORY, ALGEBRA, and PHILOSOPHY from the library COURSES and includes them in the executable image SCHEDULE.EXE.

```
$ LINK SCHEDULE,COURSES/LIBRARY/INCLUDE=(HISTORY,ALGEBRA,PHILOSOPHY)
```

This example also causes the linker to include the modules HISTORY, ALGEBRA, and PHILOSOPHY in the image file SCHEDULE.EXE. However, the /LIBRARY qualifier causes the linker to search the rest of the library COURSES and link in any other modules needed to resolve symbolic references in SCHEDULE, HISTORY, ALGEBRA, and PHILOSOPHY.

```
$ LINK UPDATE/EXE=[PROJECT.EXE]/MAP=[PROJECT.MAP]
```

This command produces the files [PROJECT.EXE]UPDATE.EXE and [PROJECT.MAP]UPDATE.MAP.

For More Information:

- On debugging (Section 8.1)
- On error messages (Section 1.4)
- On including modules from object module libraries (Section 1.2.2)
- On messages generated by the linker (*VMS Linker Utility Manual*)
- On DCL syntax (HELP or the *VMS DCL Concepts Manual*)
- On the VMS Linker (*VMS Linker Utility Manual*)

1.2.1 LINK Qualifiers

The following are command and file qualifiers that you can use when linking object modules.

/[NO]DEBUG
/NODEBUG (default)
Indicates that the VMS Debugger is to be included in the executable image and that a symbol table is to be generated. If you specify LINK/DEBUG, the program links and then executes under the control of the debugger.

/[NO]EXECUTABLE
/EXECUTABLE (default)
Produces an executable image. A file specification can follow /EXECUTABLE to designate a name for the image file. The /NOEXECUTABLE qualifier, which suppresses production of the image file, is useful when you want to verify the results of linking an object file before the image is produced.

/INCLUDE
none (default)
Specifies that the input file is an object module or a shareable image library, and that the modules named are the only ones in the library to be explicitly included as input. In the case of shareable image libraries, the module is the shareable image name. You must specify at least one module name with the /INCLUDE qualifier. The default for library modules is the .OLB file type.

This qualifier is a file qualifier and cannot be used directly on the LINK command.

/LIBRARY
none (default)
Specifies that the input file is an object module or shareable image library, which the linker must search to resolve undefined symbols within other input modules specified on the same command line.

You can use the /LIBRARY qualifier with the /INCLUDE qualifier to modify the same input file specification. In that case, the same library is searched for unresolved references.

This qualifier is a file qualifier and cannot be used directly on the LINK command. The default for the file to which this qualifier is applied is the .OLB file type.

/[NO]MAP
/NOMAP (interactive default)
/MAP/NOCROSS_REFERENCE (batch default)
Controls the generation of a map file and its contents. The /MAP qualifier produces a map file, which you can name by including a file specification.

The map file is stored on the default device in the default directory. If you do not include a file specification with /MAP, the map file is given the name of the first input file and a .MAP file type.

With the /MAP qualifier, you can use the qualifiers /BRIEF, /FULL, and /CROSS_REFERENCE to define the type of information included in the map file.

filename/OPTIONS
none (default)
Specifies that the input file is a linker options file, which can contain input file specifications as well as special instructions recognized only by the linker. You can also use options files to create shareable images.

/[NO]SHAREABLE
/NOSHAREABLE (default)
Creates a shareable image. A shareable image has all of its internal references resolved, but must be linked with one or more object modules to produce an executable image. For example, a shareable image can contain a library of routines or can be used by the system manager to create a global section for all users.

To include a shareable image as input to the linker, you can insert the shareable image into a shareable image library and specify the library as input to the LINK command. By default, the linker automatically searches the system-supplied shareable image library SYS$LIBRARY:IMAGELIB.OLB after searching any libraries you specify on the LINK command line. You can also include a shareable image by using a linker options file.

The /NOSHAREABLE qualifier specifies that the image produced cannot be linked with other images.

/[NO]TRACEBACK
/TRACEBACK (default)
Causes the generation of error messages to be accompanied by symbolic traceback information. This information shows the sequence of calls that transferred control to the program in which the error occurred. /NOTRACEBACK suppresses production of traceback information.

The traceback capability is automatically included with the /DEBUG qualifier; therefore, if you specify both /DEBUG and /NOTRACEBACK, /NOTRACEBACK has no effect.

For More Information:

- On debugging (Section 8.1)
- On object-module libraries (Section 1.2.2)
- On shareable images, options files, and contents of map files (*VMS Linker Utility Manual*)

1.2.2 Object Module Libraries

An object module library contains modules of program text that have been successfully compiled. To link modules contained in a object module library, use the /INCLUDE qualifier and specify the modules you want to link. For example:

```
$ LINK GARDEN,VEGETABLES/INCLUDE=(EGGPLANT,TOMATO,BROCCOLI,ONION)
```

This example directs the linker to link the subprogram modules EGGPLANT, TOMATO, BROCCOLI, and ONION with the main program module GARDEN.

Besides program modules, an object module library can also contain a symbol table with the names of each global symbol in the library, and the name of the module in which they are defined. You specify the name of the object module library containing symbol definitions with the /LIBRARY qualifier. When you use the /LIBRARY qualifier during a link operation, the linker searches the specified library for all unresolved references found in the included modules during compilation.

Also, by default, the linker automatically searches the system-supplied shareable image library SYS$LIBRARY:IMAGELIB.OLB after searching any libraries you specify on the LINK command.

In the following example, the linker uses the library RACQUETS to resolve undefined symbols in BADMINTON, TENNIS, and RACQUETBALL.

```
$ LINK BADMINTON, TENNIS, RACQUETBALL, RACQUETS/LIBRARY
```

You can define an object module library to be your default library by using the DCL command DEFINE. The linker searches default user libraries for unresolved references after it searches modules and libraries specified in the LINK command.

For More Information:

- On the LINK command and qualifiers (Section 1.2)
- On the VMS Linker (*VMS Linker Utility Manual*)
- On the DEFINE DCL command (*VMS DCL Dictionary*)

1.3 RUN Command

The RUN command executes programs that have been linked into an executable image by the VMS Linker.

RUN [/command-qualifier] file-spec

/command-qualifier
The name of a qualifier that indicates special processing to be performed by the linker on all files listed.

file-spec
The name of the executable image you want to run. The default file type for executable images is .EXE.

The image activator accepts one command qualifier, as follows:

/[NO]DEBUG
depends on linking (default)
The /[NO]DEBUG qualifier is optional. Specify the /DEBUG qualifier to request the debugger, if the image was not linked with it. You cannot use /DEBUG on images linked with the /NOTRACEBACK qualifier. If the image was linked with the /DEBUG qualifier and you do not want the debugger to prompt, use the /NODEBUG qualifier. The default action depends on whether you specified /DEBUG on the LINK command line.

Consider the following examples:

```
$ RUN PROG
```

This example executes the image PROG.EXE. If you specified /DEBUG to the linker while creating PROG.EXE, the image activator passes control to the debugger upon execution. If you did not specify /DEBUG to the linker while creating PROG.EXE, the image activator executes the program.

```
$ RUN/NODEBUG PROG
```

This example executes the image PROG.EXE without invoking the debugger.

For More Information:

- On debugging (Section 8.1)
- On messages generated by the image activator (Section 1.4)
- On the RUN DCL command (*VMS DCL Dictionary*)

1.4 Error Messages

During program development, you may have to respond to messages regarding possible syntax or logic errors in your program. These messages have the following form:

```
%SOURCE-CLASS-MNEMONIC, message_text
```

SOURCE
A code that identifies the origin of the message. For example, the PASCAL code identifies the VAX Pascal compiler, the LINK code identifies the VMS Linker Utility, and the PAS code identifies the VAX Pascal run-time system.

CLASS
A single character that determines message severity. The four classes of error messages are: Informational (I), Warning (W), Error (E), and Fatal (F). The definition for each class depends on the source of the message, but execution of your request does not continue when E- or F-level errors occur.

MNEMONIC
A name that is unique to that message.

message_text
Explains the event that caused the message to be generated.

For example, a common linker error occurs when you omit required file or library names from the command line, and the linker cannot locate the definition for a specified global symbol reference. The following error messages appear when a main program in OCEAN.OBJ calls a subprogram in SEAWEED.OBJ that is not specified in the LINK command:

```
%LINK-W-NUDFSYMS, 1 undefined symbol
%LINK-I-UDFSYMS,        SEAWEED
%LINK-W-USEUNDEF, module "OCEAN" references undefined symbol "SEAWEED"
%LINK-W-DIAGISUED, completed but with diagnostics
```

For More Information:

- On the complete list of VAX Pascal compile-time and run-time errors (Appendix B)
- On the complete list of linker messages (*VMS System Messages and Recovery Procedures Reference Manual*)

Chapter 2

Data Storage and Representation

This chapter describes how the VAX Pascal compiler allocates and represents program components in the VMS environment. It discusses the following topics:

- Program sections (Section 2.1)
- Storage allocation (Section 2.2)
- Allocation sizes of variables (Section 2.3)
- Alignment boundaries for variables (Section 2.4)
- Ranges of integer and real types (Section 2.5)
- Representation of varying data (Section 2.6)
- Representation of floating-point data (Section 2.7)
- Representation of schema types and variables (Section 2.8)

2.1 Program Sections

The VAX Pascal compiler uses contiguous areas of memory, called program sections, to store information about a program. The VMS Linker controls memory allocation and sharing according to the properties of each program section. (The *VMS Linker Utility Manual* refers to the various characteristics of program sections as "attributes." This chapter uses the term "properties" to avoid confusion with the VAX Pascal attribute classes.) Table 2–1 lists the possible program section properties.

Table 2–1: Program Section Properties

Property	Description
PIC/NOPIC	Position independent or position dependent
CON/OVR	Concatenated or overlaid
REL/ABS	Relocatable or absolute
GBL/LCL	Global or local scope
EXE/NOEXE	Executable or nonexecutable
RD/NORD	Readable or nonreadable
WRT/NOWRT	Writable or nonwritable
SHR/NOSHR	Shareable or nonshareable

When constructing an executable image, the linker divides the image into sections. Each image section contains program sections that have the same properties. The linker controls memory allocation by arranging image sections according to program section properties. You can use special linker options to change program section properties and to influence the memory allocation in the image. You include these options in a linker options file, which is input to the linker.

For More Information:

For information on program sections and linker options, see the *VMS Linker Utility Manual*.

2.1.1 Establishing Program Sections

If necessary, the VAX Pascal compiler can establish up to four program sections, $CODE, $LOCAL, PAS$GLOBAL, and LIB$INITIALIZE. Table 2–2 summarizes the kinds of data contained in these sections by default.

Table 2–2: Program Section Data

Program Section	Data
$CODE	Machine instructions; constants needing storage; nonvolatile, read-only, and static variables

(continued on next page)

Table 2–2 (Cont.): Program Section Data

Program Section	Data
$LOCAL	Nonexternal static types; writable variables declared with the STATIC attribute; writable variables that use default allocation and are declared at program or module level of a nonoverlaid compilation unit
PAS$GLOBAL	Writable variables that use default allocation and are declared at program or module level of an overlaid compilation unit
LIB$INITIALIZE	Addresses of routines declared with the INITIALIZE attribute and compiler-generated routines to perform module initialization

You can also establish user-defined program sections with the VAX Pascal PSECT and COMMON attributes. The PSECT attribute directs the compiler to establish a separate program section for static variables or executable blocks. In this way, you can ensure particular program section properties for these objects. You can also choose to group them with related static variables and blocks to reduce the amount of paging overhead.

The COMMON attribute directs the compiler to establish a particular program section called a common block. A common block is an overlaid program section that contains one variable. By storing variables in common blocks, a VAX Pascal program can share variables with programs written in other VAX languages.

The following example uses a common block to pass information between VAX Pascal and VAX FORTRAN:

VAX Pascal Program:

```
PROGRAM Common_Example (OUTPUT);
VAR
   Myrec  : [COMMON(Example)] RECORD
      Intfld : INTEGER;
      Strfld : PACKED ARRAY [1..10] OF CHAR;
   END;
[EXTERNAL] PROCEDURE Call_Fort; EXTERNAL;

BEGIN
Myrec := ZERO;
Call_Fort;
WRITELN('Intfld = ',Myrec.Intfld);
WRITELN('Strfld = ',Myrec.Strfld);
END.
```

VAX FORTRAN Subroutine:

```
      SUBROUTINE CALL_FORT
C
      STRUCTURE /TEST/
          INTEGER*4 ITEM
          CHARACTER * 10 ITEM_NAME
      END STRUCTURE
C
      RECORD /TEST/ VAR
      COMMON /EXAMPLE/ VAR
C
      VAR.ITEM = 10
      VAR.ITEM_NAME = '0123456789'
      END
```

The VAX Pascal program initializes the common record, Myrec, to zero, then calls the VAX FORTRAN routine, Call_Fort, to assign the desired values to the elements within the common record. The VAX Pascal program then writes the assigned values to SYS$OUTPUT.

Note that only one variable can be allocated in a particular VAX Pascal common block. To share more than one data item in the same common block, the record variable containing all shareable items is declared and used.

For More Information:

For information on the PSECT and COMMON attributes, see the *VAX Pascal Reference Manual.*

2.1.2 Establishing Program Section Properties

Whether the compiler establishes a program section, or you create one, the program section is assigned one property from each class listed in Table 2–1. These properties are assigned to satisfy the requirements of the types, variables, and executable blocks that have been allocated in the same program section. Table 2–3 lists the minimal properties required by various objects.

Table 2–3: Required Program Section Properties

Object	Properties			
	EXE	RD	WRT	NOSHR
Read-only Variable		X		
Write-only Variable		X	X	X[1]
Read/write Variable		X	X	X[1]
Executable Block	X	X		

[1]Unless the variable has the COMMON attribute

The . ABS . program section properties are ABS, NOEXE, NORD, NOWRT, and NOSHR. All other program sections except LIB$INITIALIZE are position independent and relocatable; all except those established by the COMMON attribute are concatenated and local. Program sections established by COMMON are overlaid and global. The remaining properties are assigned as follows:

- The first time the compiler encounters the name of a particular program section (including a common block), it initializes the program section to be readable, nonwritable, shareable, and nonexecutable. Thus, the program section's initial properties are LCL, NOEXE, NOWRT, CON, PIC, RD, REL, and SHR.
- If storage for any writable object not in a common block is allocated in the same program section, the program section instantly becomes writable and nonshareable. Thus, the program section's write property changes from NOWRT to WRT, and its share property changes from SHR to NOSHR.
- If storage for a writable object in a common block is allocated in the same program section, the program section becomes writable and retains the shareable property. Thus, the program section's write property changes from NOWRT to WRT, and its share property remains SHR.

If you want to guarantee that read-only variables can never be modified, you can allocate storage exclusively for them in a separate program section that will always remain nonwritable.

2.2 Storage Allocation

The following sections discuss storage allocation of variables, symbolic constants, and executable blocks. Static types require no allocation. Nonstatic data types require allocation to store possible run-time values. Section 2.2.3 gives an example.

For More Information:

For information on nonstatic data types, see Section 2.8.

2.2.1 Allocation of Variables

When allocating storage for a variable, the compiler first determines an allocation attribute for the variable. If the allocation attribute is STATIC or COMMON, the compiler then chooses a program section in which to allocate storage. The compiler applies the following rules sequentially to determine the allocation attribute:

- If the variable is declared with an allocation attribute, the attribute specifies the variable's allocation.
- If the variable is declared with a visibility attribute other than LOCAL, its allocation is static.
- If the variable is declared in a routine, its allocation is automatic.
- If the variable is declared at the outermost level of a module, its allocation is static.
- If the variable is declared at the outermost level of a program, the compiler must choose between static and automatic allocation. Whenever possible, the compiler uses automatic allocation for variables that are referred to only in the body of the main program because automatic allocation is more efficient. The compiler uses static allocation if the variable is declared with the VOLATILE attribute, initialized at its declaration, referred to in a nested block, or if the program has an ENVIRONMENT or OVERLAID attribute. Because program-level variables may be statically allocated, VAX Pascal does not support recursive calls on the main program block.

The compiler applies the following rules sequentially to choose the program section in which to allocate storage for nonexternal common and static variables:

- If the variable has the COMMON attribute, storage is allocated in a common block that has either the same name as the variable, or the name specified by the identifier that accompanies the attribute.
- If the variable has the PSECT attribute, the identifier that accompanies the attribute supplies the name of the program section in which storage is to be allocated.
- If the variable does not have the COMMON, PSECT, or STATIC attribute, but is declared at the outermost level of an overlaid compilation unit, storage is allocated in the program section PAS$GLOBAL.
- If the variable has the READONLY attribute and has neither a PSECT nor a COMMON attribute, its storage is allocated in the program section in which storage for executable code is currently being allocated.
- All other static variables are allocated in the program section $LOCAL.

For More Information:

For information on attributes, see the *VAX Pascal Reference Manual*.

2.2.2 Allocation of Symbolic Constants and Executable Blocks

When allocating storage for symbolic constants and executable blocks, the compiler determines the appropriate program section by applying the same rules of scope to program section names that it applies to identifiers. The compiler always allocates storage in the program section whose name appeared in the most recent heading of a routine or compilation unit.

When a program begins, the compiler establishes the $CODE program section. Storage is allocated in $CODE for symbolic constants and executable blocks unless a PSECT attribute appears in the heading of a routine or compilation unit and directs that storage be allocated in a different program section.

If a routine has the INITIALIZE attribute, the address of its entry mask is stored in a program section named LIB$INITIALIZE.

For More Information:

- On the scope of VAX Pascal identifiers, the PSECT attribute, and the INITIALIZE attribute (*VAX Pascal Reference Manual*)
- On LIB$INITIALIZE (*VMS Run-Time Library Routines Volume*)

2.2.3 Allocation Example

Example 2–1 illustrates how the compiler establishes program sections to allocate storage for symbolic constants, variables, and executable blocks. The comments in the programs indicate the names of the program sections used.

Example 2–1: Using Program Sections to Allocate Storage

```
PROGRAM Allocate_Variables (INPUT,OUTPUT);

CONST
   Message_String = 'Random String Literal';        { $CODE }

VAR
   Magic_Number : [READONLY,PSECT(Magic)] INTEGER
                        VALUE 42;                   { Magic }
   Local_Variable : INTEGER;                        { $LOCAL }

[PSECT(Error_Routines)] PROCEDURE User_Error;

   CONST
      User_Error_Message = 'Internal Error';        { Error_Routines }

   VAR
      Error_Count : [STATIC] INTEGER VALUE 0;       { $LOCAL }
      Message_Buffer : VARYING [132] OF CHAR;       { Automatic Storage }

   BEGIN { Error_Routines }
   Error_Count := Error_Count + 1;
   Local_Variable := Error_Count;
   END;

BEGIN                                               { $CODE }
   .
   .
   .
END.
```

Storage for all variables with static allocation is allocated in $LOCAL. Storage for the executable block of the main program and the symbolic constant Message_String is allocated in $CODE. Storage is allocated in the user-created program section, Error_Routines, for the symbolic constant User_Error_Message and the executable block of User_Error. Storage for Local_Variable is in $LOCAL because it was referred to in a nested block. Storage for Magic_Number is in the user-created program section, Magic. Since Magic_Number was declared with the READONLY attribute, the program section has the NOEXE, NOWRT, RD, and SHR properties.

2.3 Allocation Sizes of Variables

For every VAX Pascal data type, the compiler calculates the allocation size required when a variable of the type occurs in either an unpacked or a packed context. The unpacked size is always represented in bytes, while the packed size is represented in bits.

The packed size of a variable is the minimum number of bits required to represent all values of the variable's type. In general, the compiler uses the following 32–bit rule to determine the allocation size for a component of a structured variable:

- A component whose length is 32 bits or fewer is packed into as few bits as possible and can be unaligned.
- A component whose length is greater than 32 bits is allocated the smallest number of bytes possible and must be aligned on a byte boundary.

If one of the size attributes (BIT, BYTE, WORD, LONG, QUAD, or OCTA) is applied to the variable, the size specified by the attribute represents the variable's packed size. If no size attribute is applied to the variable, the compiler calculates the unpacked size so that it is structurally compatible with the base type of the variable's type.

Table 2–4 illustrates the allocation size for variables of each type when they occur in either a packed or an unpacked context.

Table 2–4: Storage of Types

Data Type	Unpacked Size in Bytes	Packed Size in Bits
INTEGER	4	32
UNSIGNED	4	32
CHAR	1	8
BOOLEAN	1	1
Enumerated[2]	1, if 256 elements or fewer; 2, if more than 256 elements	log2(number of elements)[1] + 1
Subrange	The size of the base type	If either the upper or lower bounds contain a reference to a formal discriminant, then the size of the base type, otherwise the minimum number in which the upper and lower bounds can be expressed[3]
REAL or SINGLE	4	32
DOUBLE	8	64
QUADRUPLE	16	128
Pointer	4	32
Unpacked ARRAY	The sum of the unpacked sizes in bytes of all components, plus the sum of the sizes in bytes of any holes created to meet alignment requirements	Unpacked size in bytes * 8

[1]This is known as the ceiling function, where the smallest integer is greater than or equal to X.

[2]The maximum number of elements is 65,535.

[3]Sets of type INTEGER and UNSIGNED are limited to 256 bits.

(continued on next page)

Table 2–4 (Cont.): Storage of Types

Data Type	Unpacked Size in Bytes	Packed Size in Bits
Unpacked RECORD	The sum of the unpacked sizes in bytes of the fields in the fixed part and the largest variant, plus the sum of the sizes in bytes of any holes created to meet alignment requirements	Unpacked size in bytes * 8
PACKED ARRAY	The sum of the packed sizes in bits of all components, plus the sum of sizes in bits of any holes created to meet alignment requirements; this sum is rounded up to a multiple of 8 and then divided by 8	The sum of the packed sizes in bits of all components, plus the sum of sizes in bits of any holes created to meet alignment requirements; if the sum is greater than 32, the sum is rounded up to the next multiple of 8
PACKED RECORD	The sum of the packed sizes in bits of all fields in the fixed part and the largest variant, plus the sum of sizes in bits of any holes created to meet alignment requirements; this sum is rounded up to a multiple of 8 and then divided by 8	The sum of the packed sizes in bits of all fields in the fixed part and the largest variant, plus the sum of sizes in bits of any holes created to meet alignment requirements; if the sum is greater than 32, the sum is rounded up to the next multiple of 8
STRING, VARYING OF CHAR	Maximum length + 2	(Maximum length + 2) * 8
Nonstatic PACKED SET, Unpacked SET[3]	32, if the set base type is subrange of INTEGER or UNSIGNED; else, compute (ORD(upper-bound of ordinal type that is base type of set's base type) + 8) DIV 8, and if this result is less than or equal to 8, the result is rounded up to 1, 2, 4, or 8	Compute (ORD (upper-bound) + 1); if the result is less than or equal to 64, the result is rounded up to 8, 16, 32, or 64, and if the result is greater than 64, the result is rounded to next higher multiple of 8

[3]Sets of type INTEGER and UNSIGNED are limited to 256 bits.

(continued on next page)

Table 2–4 (Cont.): Storage of Types

Data Type	Unpacked Size in Bytes	Packed Size in Bits
PACKED SET[3]	The result of (ORD(upper-bound) + 8) DIV 8	Compute (ORD (upper-bound) + 1); if the result is greater than 32, the result is rounded to next higher multiple of 8
FILE	Not specified	Not specified

[3]Sets of type INTEGER and UNSIGNED are limited to 256 bits.

The formula to compute the size of a packed subrange is MAX (X,Y) + Z where X, Y, and Z are computed as follows (LOW represents the low bound of the subrange; HIGH represents the upper bound):

```
IF LOW < --1
THEN
   X := log2(-LOW - 1) + 1
ELSE
   X := 0;

IF LOW >= 0
THEN
   Z := 0
ELSE
   Z := 1;
```

```
 IF HIGH> 0
THEN
   Y := log2(HIGH) + 1
ELSE
   Y := 0;
```

You can discover both the unpacked and packed sizes for variables of any type by using the predeclared functions BITNEXT, BITSIZE, NEXT, and SIZE, which require a parameter that is the name of either a type or a variable. These functions return the following integer values:

- BITNEXT returns an integer value that indicates what the packed size would be for an array component of the type.
- BITSIZE returns an integer value that indicates what the packed size would be for a record field of the type.
- NEXT returns an integer value that indicates what the unpacked size would be for an array component of the type.
- SIZE returns an integer value that indicates what the unpacked size would be for a variable or record field of the type.

For More Information:

- On size attributes (*VAX Pascal Reference Manual*)
- On predeclared functions (*VAX Pascal Reference Manual*)

2.3.1 Allocation Size Examples

The following examples illustrate the effects of packing records and multidimensional arrays at various levels.

Example 1

```
TYPE
   Internal_Arr = ARRAY[1..5] of 0..6;

VAR
   Samp1_Arr : PACKED ARRAY[1..5] OF Internal_Arr;
```

Each component of an array of type Internal_Arr is stored in a longword. Each component of Samp1_Arr, in turn, requires five longwords, which is enough storage space for five components of type Internal_Arr. The entire array Samp1_Arr therefore occupies 25 longwords (800 bits).

Example 2

```
VAR
   Samp1_Arr : ARRAY[1..5] OF PACKED ARRAY[1..5] OF 0..6;
```

Each PACKED ARRAY[1..5] of 0..6 requires 15 bits. Because the packed arrays are components of an unpacked array, their size is rounded up to an even 16 bits. Therefore, the total size of Samp1_Arr is 80 bits.

Example 3

```
TYPE
   Internal_Arr = PACKED ARRAY[1..5] OF 0..6;

VAR
   Samp2_Arr : PACKED ARRAY[1..5] OF Internal_Arr;
   Samp3_Arr : PACKED ARRAY[1..5,1..5] OF 0..6;
```

In this example, every component of Internal_Arr requires only three bits because the array is packed. Each component of Samp2_Arr and Samp3_Arr can be stored in 15 bits, and each array occupies 75 bits. Except when the program is compiled with the /OLD_VERSION qualifier, the specification of PACKED for an array with multiple indexes results in packing at every level. Therefore, the two arrays in this example are equivalent.

Example 4

```
VAR
   Sample : PACKED ARRAY[1..5,1..5,1..5] OF 0..6;
```

This example shows space savings for arrays of more than two dimensions when PACKED is specified at every level. The subrange 0..6 requires 3 bits; five 3-bit components require 15 bits. This size describes the innermost dimension of Sample. Next, five 15-bit components require 75 bits. Because of the 32-bit rule, each 75-bit component is rounded up to 80 bits. This size describes the middle and inner dimensions of Sample. Finally, five 80-bit components require 400 bits (50 bytes). The entire array Sample, then, requires 400 bits.

Example 5

```
VAR
   Sample_Rec : PACKED RECORD
                   Field_1 : BOOLEAN;
                   Field_2 : INTEGER;
                   Field_3 : DOUBLE;
                END;
```

In this example, Field_1 requires only 1 bit of storage. Field_2 is 32 bits in size and starts immediately following Field_1. Because Field_3 is larger than 32 bits, it will start on the next byte boundary. The entire record Sample_Rec, therefore, requires 104 bits.

2.4 Alignment Boundaries

The memory-addressing boundary on which a variable or a component is aligned depends on the variable's or the component's allocation size. You can change the alignment by using the ALIGNED and UNALIGNED attributes. The following conditions determine boundary alignment:

- If the variable or component has an alignment attribute, the compiler uses the specified alignment.
- Variables declared without an alignment attribute are aligned by default on a byte boundary. The compiler can align such variables on a larger storage boundary if it can access them more efficiently by doing so.
- All components of an unpacked array or record variable are byte aligned within the array or record. The array or record itself, however, may be unaligned.
- The alignment of components of a packed array, packed record, or VARYING OF CHAR variable always follows the 32-bit rule.
- Dynamic variables allocated by the NEW procedure are always aligned on a quadword boundary.

Although you can save storage space by packing variables of structured types, you must be careful to pack at the proper level. Except for its alignment, a record field whose type is an unpacked array, set, or record occupies the same amount of space in a packed or an unpacked record variable. To pack such a field, you must explicitly declare its type to be packed.

When packing multidimensional arrays, you must specify packing at the innermost level to gain any significant space advantage. For example, there is no advantage in packing an array of an unpacked structured type; the unpacked components will still be aligned on byte boundaries, thereby leaving holes in the storage space. To gain storage space, you must specify a packed array of a packed structured type.

For More Information:

- On the 32-bit rule (Section 2.3)
- On the ALIGNED and UNALIGNED attributes (*VAX Pascal Reference Manual*)

2.4.1 Allignment Boundary Examples

The following examples show the use of alignment boundaries.

Example 1

```
VAR
   X : [STATIC, ALIGNED (3)] INTEGER;  { $LOCAL, QUADWORD ALIGNED }
```

In this example, X is declared a static variable that is aligned on a QUADWORD boundary.

Example 2

```
VAR
   Page_Name : [PSECT(New_Section), ALIGNED(9)] PACKED ARRAY [1..512] OF
      [BYTE] 0..255;              { New_Section, PAGE ALIGNED }
```

This example declares the page aligned variable Page_Name, which is to be allocated in the program section New_Section.

Example 3

```
VAR
   X : PACKED RECORD                    { AUTOMATIC }
          Field1 : BOOLEAN;
          Field2 : REAL;
          Field3 : BOOLEAN;
          Field4 : DOUBLE;
       END;
```

In this example, Field1 begins at bit position 0 and is 1 bit long. Field2 begins at bit position 1 and is 32 bits long. Field3 begins at bit position 33 and is 1 bit long. However, because Field4 is greater than 32 bits long (DOUBLE requires 64 bits) Field4 must be byte aligned. For this reason, Field4 begins on bit position 40 and is 64 bits long.

Example 4

```
VAR
   X : RECORD                           { AUTOMATIC }
          Field1 : [BIT(3)]  0..7;
          Field2 : [UNALIGNED] INTEGER;
       END;
```

In this example, Field1 of record X is declared to begin on bit position 0 and is 3 bits long. Due to the use of the UNALIGNED attribute on Field2, Field2 begins on bit position 3 and is 32 bits long. Note that you can obtain the same behavior by packing record X.

Without using the UNALIGNED attribute, or without X being a PACKED record, Field2 would have been byte aligned; that is, it would have started on bit position 8.

2.5 Ranges of Integer and Real types

The VAX Pascal compiler assigns a range of numeric values to integer and real data types.

The range of values for integer types are shown in Table 2–5.

Table 2–5: Range of Integer Types

Integer Type	Range of Values	Number Range
INTEGER[1]	$-2^{31}+1$ through $2^{31}-1$	–2,147,483,647 through 2,147,483,647
UNSIGNED[2]	0 through $2^{32}-1$	0 through 4,294,967,295[3]

[1]Predefined identifier range is –MAXINT through MAXINT.
[2]Predefined identifier for the highest value in the range is MAXUNSIGNED.
[3]All nonnegative integer values.

The range of values for real types are shown in Table 2–6.

Table 2–6: Range of Real Types

Real Type	Degree of Precision	Range of Values
REAL(SINGLE)	Single-precision	$0.29 * (10^{38})$ through $1.7 * (10^{38})$
DOUBLE	Double-precision	
	D_floating	$0.29 * (10^{38})$ through $1.7 * (10^{38})$
	G_floating	$0.56 * (10^{-308})$ through $0.90 * (10^{308})$
QUADRUPLE	Quadruple-precision	$0.84 * (10^{-4932})$ through $0.59 * (10^{4932})$

For More Information:

- On integer and real types (*VAX Pascal Reference Manual*)
- On floating-point representation (Section 2.7)
- On predeclared identifiers for the range of positive values in real types (*VAX Pascal Reference Manual*)
- On predeclared identifiers representing the precision of floating-point types around 1.0 (*VAX Pascal Reference Manual*)

2.6 Representation of Varying Data

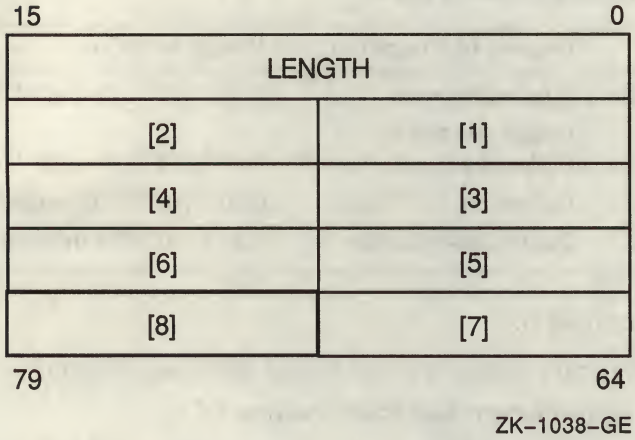
This section summarizes the internal representation of VARYING OF CHAR types. A variable of type VARYING OF CHAR is stored as though it were a VAX Pascal record type of the following form:

```
RECORD
   Length : [WORD] 0..Maxlength;
   Body : PACKED ARRAY[1..Maxlength] OF CHAR;
END;
```

Storage is allocated as one byte per character, with an initial field of two bytes to indicate the total length. A variable of type VARYING OF CHAR whose length is greater than or equal to three characters is allocated an exact number of bytes. Storage allocation for a variable of type VARYING OF CHAR whose maximum length is zero, one, or two characters follows the 32-bit rule in that the variable can have the UNALIGNED attribute.

Figure 2–1 illustrates the storage allocated for a variable of type VARYING[8] OF CHAR.

Figure 2–1: Storage of Varying Data



The predefined schema type STRING uses VARYING OF CHAR as its underlying data type, therefore Figure 2–1 also represents the STRING type.

For More Information:

- On the 32-bit rule (Section 2.3)
- On VAX Pascal varying data types (*VAX Pascal Reference Manual*)
- On the UNALIGNED attribute (*VAX Pascal Reference Manual*)
- On VAX Pascal string data types (*VAX Pascal Reference Manual*)

2.7 Representation of Floating-Point Data

The following sections summarize the internal representation of single-precision (REAL and SINGLE types), double-precision (DOUBLE type), and quadruple-precision (QUADRUPLE type) floating-point numbers.

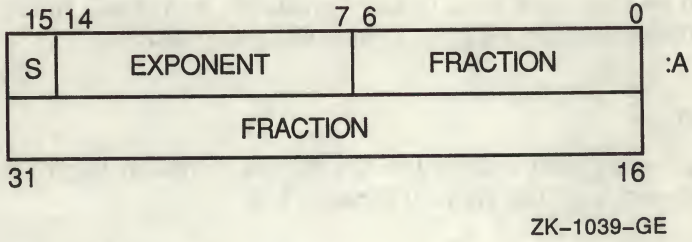
For More Information:

- On internal representation (*Introduction to VMS System Routines*)
- On VAX Pascal floating_point data types (*VAX Pascal Reference Manual*)

2.7.1 Single-Precision (SINGLE and REAL Types)

A single-precision floating-point value is represented by four contiguous bytes. The bits are numbered from the right, 0 through 31, as shown in Figure 2–2.

Figure 2–2: Single-Precision Floating-Point Data Representation



A single-precision floating-point value is specified by its address A, the address of the byte containing bit 0. The form of this value is sign magnitude as follows:

- Bit 15 is the sign bit.
- Bits 14 through 7 are an excess 128 binary exponent.
- Bits 6 through 0 and 31 through 16 are a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go from 16 through 31 and from 0 through 6.

The 8-bit exponent field encodes the values 0 through 255 as follows:

- An exponent value of 0, with a sign bit of 0, indicates that the floating-point number has a value of 0.
- Exponent values of 1 through 255 indicate binary exponents of –127 through +127.
- An exponent value of 0, with a sign bit of 1, is considered a reserved operand. Floating-point instructions that process a reserved operand cause a reserved operand fault.

The precision of a single-precision value is approximately one part in 2^{23}, or 7 decimal digits.

For More Information:

For information on the single-precision range, see Section 2.5.

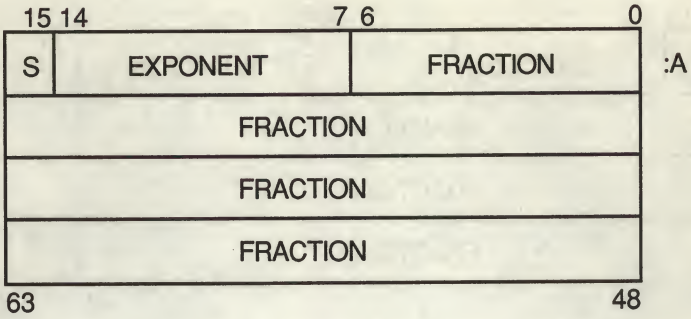
2.7.2 Double-Precision (DOUBLE Type)

A double-precision floating-point value is represented by eight contiguous bytes. Variables of type DOUBLE take one of two formats: D_floating-point or G_floating-point. Of these two formats, the D_floating-point format allows values to be expressed with greater precision, and the G_floating-point format allows a wider range of values to be expressed.

2.7.2.1 D_Floating-Point

D_floating-point data is stored in the format shown in Figure 2–3. The bits are numbered from the right, 0 through 63.

Figure 2–3: D_Floating-Point Double-Precision Representation



A D_floating-point value is specified by its address A, the address of the byte containing bit 0. The form of this value is identical to that of a single-precision floating-point value except for an additional 32 low-significance fraction bits. Within the fraction, bits of increasing significance are numbered 48 through 63, 32 through 47, 16 through 31, and 0 through 6.

The precision of a D_floating-point value is approximately one part in 2^{55}, or 16 decimal digits.

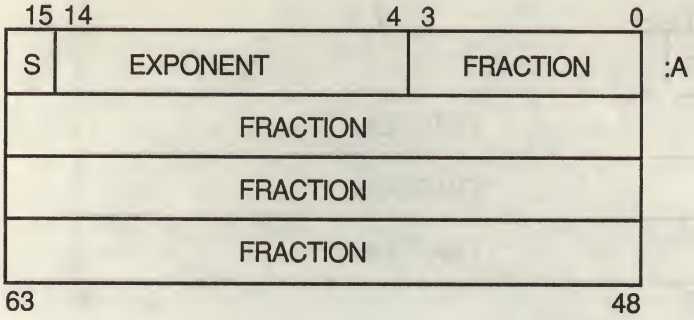
For More Information:

For information on the D-floating-point range, see Section 2.5.

2.7.2.2 G_Floating-Point

G_floating-point data is stored in the format shown in Figure 2–4. The bits are numbered from the right, 0 through 63.

Figure 2–4: G_Floating-Point Double-Precision Representation



NOTE

Not all VAX processors support G_Floating-point numbers.

A G_floating-point value is specified by its address A, the address of the byte containing bit 0. The form of this value is sign magnitude as follows:

- Bit 15 is the sign bit.
- Bits 14 through 4 are an excess 1024 binary exponent.
- Bits 3 through 0 and 63 through 16 represent a normalized 53-bit fraction without the redundant most significant fraction bit. Within the fraction, bits of increasing significance go from 48 through 63, 32 through 47, 16 through 31, and 0 through 3.

The 11-bit exponent field encodes the values 0 through 2047 as follows:

- An exponent value of 0, with a sign bit of 0, indicates that the G_floating-point number has a value of 0.
- Exponent values of 1 through 2047 indicate binary exponents of –1023 through +1023.
- An exponent value of 0, together with a sign bit of 1, is considered a reserved operand. Floating-point instructions processing a reserved operand cause a reserved operand fault.

The precision of a G_floating-point value is approximately one part in 2^{52}, or 15 decimal digits.

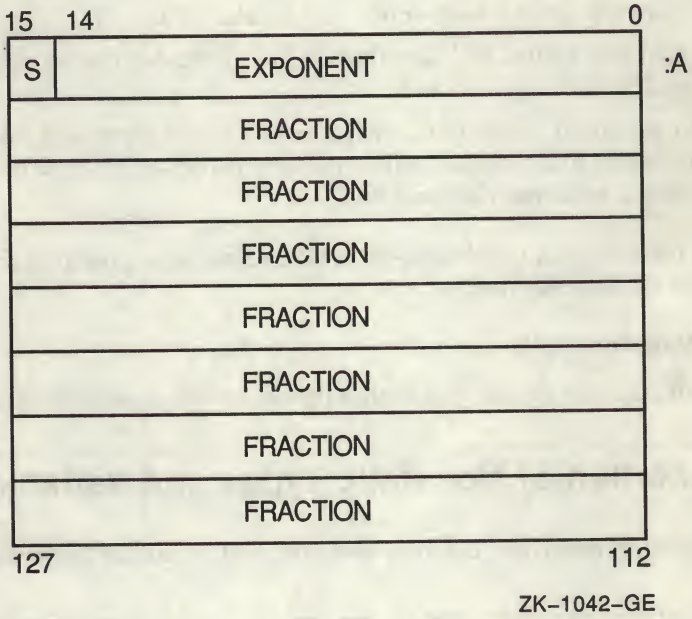
For More Information:

For information on the G_floating-point range, see Section 2.5.

2.7.3 Quadruple-Precision (QUADRUPLE Type)

A quadruple-precision floating-point value is represented by 16 contiguous bytes. The bits are numbered from the right 0 through 127, as shown in Figure 2–5.

Figure 2–5: Quadruple-Precision Floating-Point Representation



NOTE

Not all VAX processors support Quadruple (H_Floating) numbers.

A quadruple-precision floating-point value is specified by its address A, the address of the byte containing bit 0. The form of this value is sign magnitude as follows:

- Bit 15 is the sign bit.
- Bits 14 through 0 are an excess 16,384 binary exponent.
- Bits 127 through 16 are a normalized 113-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go from 112 through 127, 96 through 111, 80 through 95, 64 through 79, 48 through 63, 32 through 47, and 16 through 31.

The 15-bit exponent field encodes the values 0 through 32,767:

- An exponent value of 0, with a sign bit of 0, indicates that the quadruple-precision number has a value of 0.
- Exponent values of 1 through 32,767 indicate true binary exponents of –16,383 through +16,383.
- An exponent value of 0, with a sign bit of 1, is considered a reserved operand. Floating-point instructions processing a reserved operand cause a reserved operand fault.

The precision of a quadruple-precision value is approximately one part in 2^{112}, or 33 decimal digits.

For More Information:

For information on the quadruple-precision range, see Section 2.5.

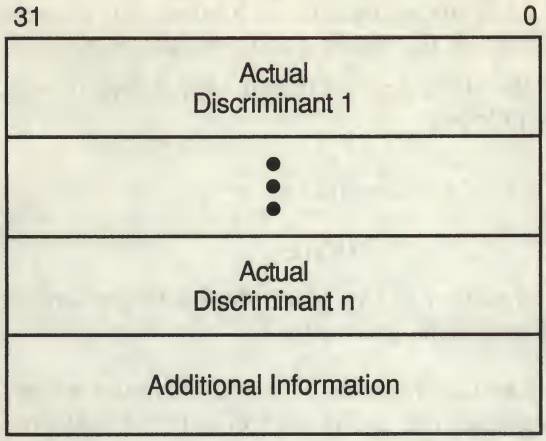
2.8 Representation of Nonstatic Types and Variables

This section describes the representation of nonstatic types and variables.

2.8.1 Representation of Nonstatic Types

Each nonstatic data type has some storage associated with it, called the control part. Figure 2–6 shows the layout of a control part of a nonstatic data type.

Figure 2–6: Storage of Nonstatic Data Types



ZK–1406A–GE

In the top portion of the control part, VAX Pascal stores each actual discriminant of the schema type in a longword of storage. The additional information piece of the control part varies in content and size depending on the type specification, and can contain any of the following:

- No information (if the schema type is simple), as follows:

```
TYPE
   A_Char( x,y : CHAR ) = x..y;
```

- Control parts of nested discriminated schema types, as follows:

```
TYPE
   My_Record( a, b : INTEGER ) = RECORD
      f1 : STRING( a );
      f2 : STRING( b );
      END;
```

- Values for all expressions appearing in the type definition, as follows:

```
TYPE
   My_Subrange( a, b : INTEGER ) = a..a+b;
```

  VAX Pascal evaluates the expression a+b when the schema type is discriminated and saves the result in the control part.

- The total size of the data part, if it can vary based on actual discriminants, as follows:

```
TYPE
   Arr( a, b : INTEGER ) = ARRAY[a..b] OF REAL;
```

NOTE

The order of information in the "additional information" section of the control part cannot be guaranteed.

If you declare more than one variable of a discriminated schema type, each variable shares the information in the control part for that type.

2.8.2 Representation of Variables of Nonstatic Types

When allocating storage for a variable of a nonstatic type, VAX Pascal allocates a pointer part and a data part. VAX Pascal allocates and initializes the pointer part (to point to the data part); you cannot access the pointer part in your program. VAX Pascal associates each object with a control part according to its data type.

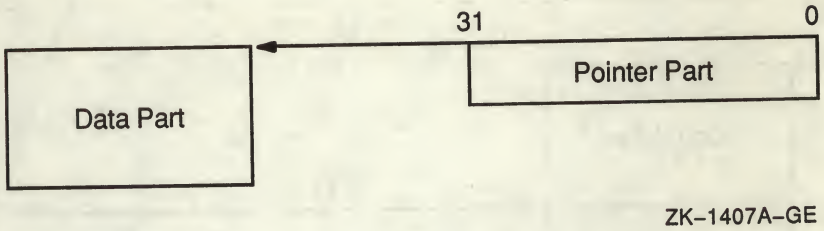
If the type of the object involves more than one nonstatic type, VAX Pascal associates that object with all applicable control parts. Consider the following:

```
TYPE
   Sub_Range( a, b : INTEGER ) = a..b;

VAR  {x requires information in two control parts:}
   x : ARRAY[Sub_Range( i, j ), Sub_Range( k, l )] OF INTEGER;
```

Figure 2–7 illustrates the layout for an object of a nonstatic type.

Figure 2–7: Storage of Variables of Nonstatic Types



In Figure 2–7, the pointer part is directly accessible by your program. The data part is allocated in heap when you use the NEW procedure, for example:

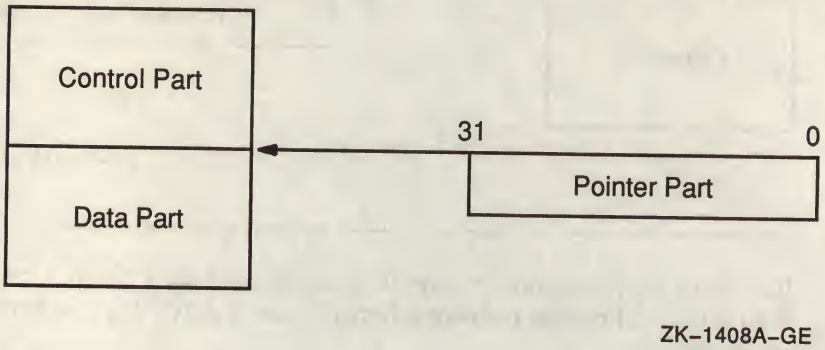
```
TYPE
   dstr = STRING( I );

VAR
   ptr : ^dstr;
{In the executable section:}
NEW( ptr );
```

You cannot create variables of undiscriminated types, but you can also show the representation for pointers to nonstatic types (except undiscriminated schema).

Figure 2–8 illustrates the layout for a pointer to an undiscriminated schema type.

Figure 2–8: Storage of Pointer Variables to Undiscriminated Schema Types



In Figure 2–8, the control part and data part are allocated together by a call to the NEW procedure. Each object allocated this way has its own control part since the base type of the pointer is undiscriminated and does not have a control part. Consider the following:

```
VAR
   x, y, z : ^STRING;
{In the executable section:}
NEW( x, 10 );    {x has a control and data part}
y := x;          {y points to same control and data part as x}
NEW( z, 10 );    {z has separate control and data parts}
```

Each variable created by NEW contains a unique control part attached to the data part.

2.8.2.1 Representation Nonstatic Record Fields

If a record object contains a field of a nonstatic type, VAX Pascal stores the field in one piece of storage within the record's storage (VAX Pascal does not create a pointer part and a data part). VAX Pascal determines the offset of the object by accessing the information in the control part of the field's data type and information in the control part of the record.

Chapter 3

Calling Conventions

This chapter describes how VAX Pascal passes parameters and calls routines in the VMS environment. It discusses the following topics:

- VAX Procedure Calling Standard (Section 3.1)
- Parameter-passing semantics (Section 3.2)
- Parameter-passing mechanisms (Section 3.3)
- Passing parameters from VAX Pascal routines to non-VAX Pascal routines (Section 3.4)
- Passing parameters from non-VAX Pascal routines to VAX Pascal routines (Section 3.5)

Note that the *Introduction to VMS System Routines* uses the term "procedure" to mean any routine entered by a CALL instruction. This chapter uses the term "routine" instead, to avoid confusion with VAX Pascal's definition of "procedure."

For More Information:

- On declaring and calling VAX Pascal routines (*VAX Pascal Reference Manual*)
- On procedure-calling and argument-passing mechanisms (*Introduction to VMS System Routines*)

3.1 VAX Procedure Calling Standard

Programs compiled by the VAX Pascal compiler conform to the VAX Procedure Calling Standard. This standard describes how parameters are passed, how function values are returned, and how routines receive and return control. By means of the calling standard, VAX Pascal provides features that allow programs to call system routines written in other native-mode languages supported by the VMS system.

For More Information:

For information on the VAX Procedure Calling Standard, see the *Introduction to VMS System Routines*.

3.1.1 Parameter Lists

Each time a routine is called, the VAX Pascal compiler constructs a parameter list. The VAX Procedure Calling Standard defines a parameter list as a sequence of longword (4-byte) entries. The first byte of the first entry in the list is a parameter count, which indicates how many parameters follow in the list.

The form in which the parameters in the list are represented is determined by the passing mechanisms you specify in the formal parameter list and the values you pass in the actual parameter list. The parameter list contains the actual parameters passed to the routine.

3.1.2 Function Return Values

In VAX Pascal, a function returns to the calling block the value that was assigned to its identifier during execution. The compiler chooses one of the following three methods for returning this value; the method chosen depends on the amount of storage required for values of the type returned:

- If the value can be represented in 32 bits of storage, it is returned in register R0.
- If the value requires from 33 to 64 bits, the low-order bits of the result are returned in register R0 and the high-order bits are returned in register R1.

- If the value is too large to be represented in 64 bits, if its type is a string type (PACKED ARRAY OF CHAR, VARYING OF CHAR, or STRING), or if the type is nonstatic, the calling routine allocates the required storage. An extra parameter (a pointer to the location where the function result will be stored) is added to the beginning of the calling routine's actual parameter list.

For values of structured types, the amount of storage required for the entire structure determines which of the three return methods is used. Character strings and objects of nonstatic types are always returned by the extra-parameter method.

Note that functions that require the use of an extra parameter can have no more than 254 parameters; functions that store their results in registers R0 and R1 can have 255 parameters.

Table 3–1 lists the methods by which values of each type are returned.

Table 3–1: Function Return Methods

Type	Return Method
INTEGER, UNSIGNED, CHAR, BOOLEAN, REAL, SINGLE, pointer, enumerated, subrange, schema subrange	General register R0
DOUBLE	R0: Low-order result R1: High-order result
QUADRUPLE, VARYING OF CHAR, PACKED ARRAY OF CHAR, STRING, other nonstatic types	Extra-parameter
Other structured types	Depends on the amount of storage required

3.1.3 Contents of the Call Stack

Each time a routine is called by a VAX Pascal program, the hardware creates a structure on the call stack; this structure is known as the call frame. The call frame for each active routine contains the following:

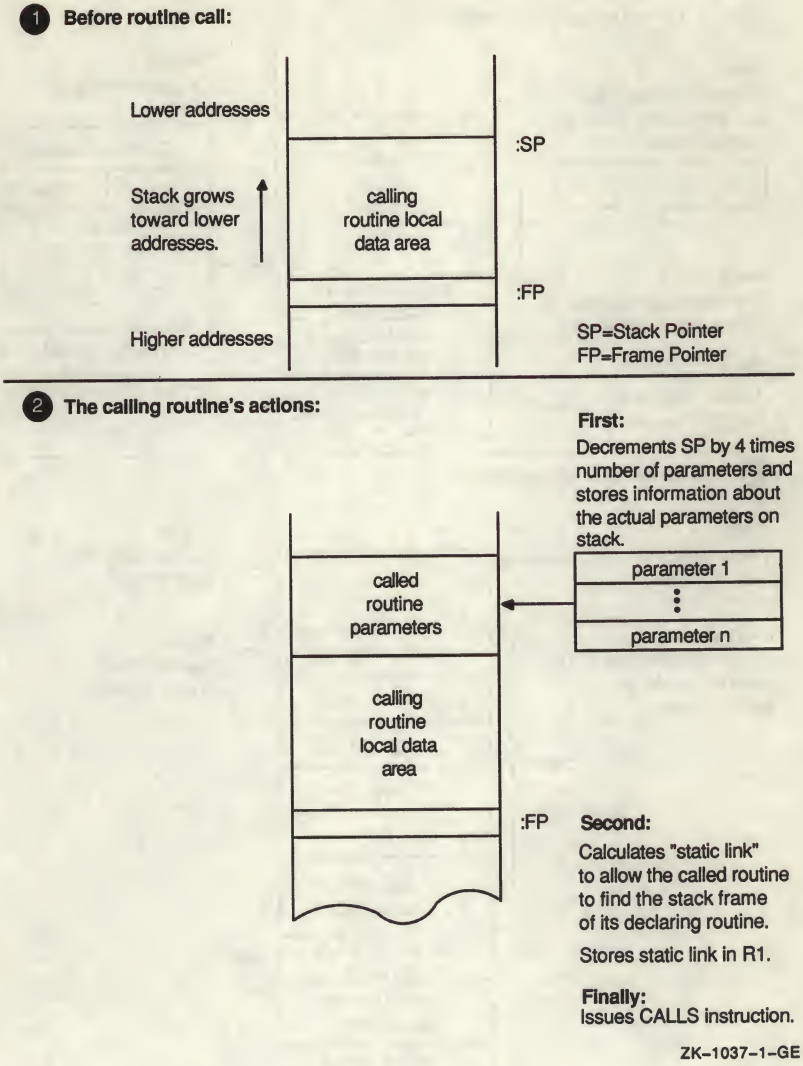
- A pointer to the call frame of the previous routine call. This pointer is called the saved frame pointer (FP).
- The saved argument pointer (AP) of the previous routine call.

- The storage address of the point at which the routine was called; that is, the address of the instruction following the call to the current routine. This address is called the saved program counter (PC).
- The saved contents of other general registers. Based on a mask specified in the control information, the system restores the saved contents of these registers to the calling routine when control returns to it.

When execution of a routine ceases, the system uses the frame pointer in the call frame of the current routine to locate the frame of the previous routine. The system then removes the call frame of the current routine from the stack.

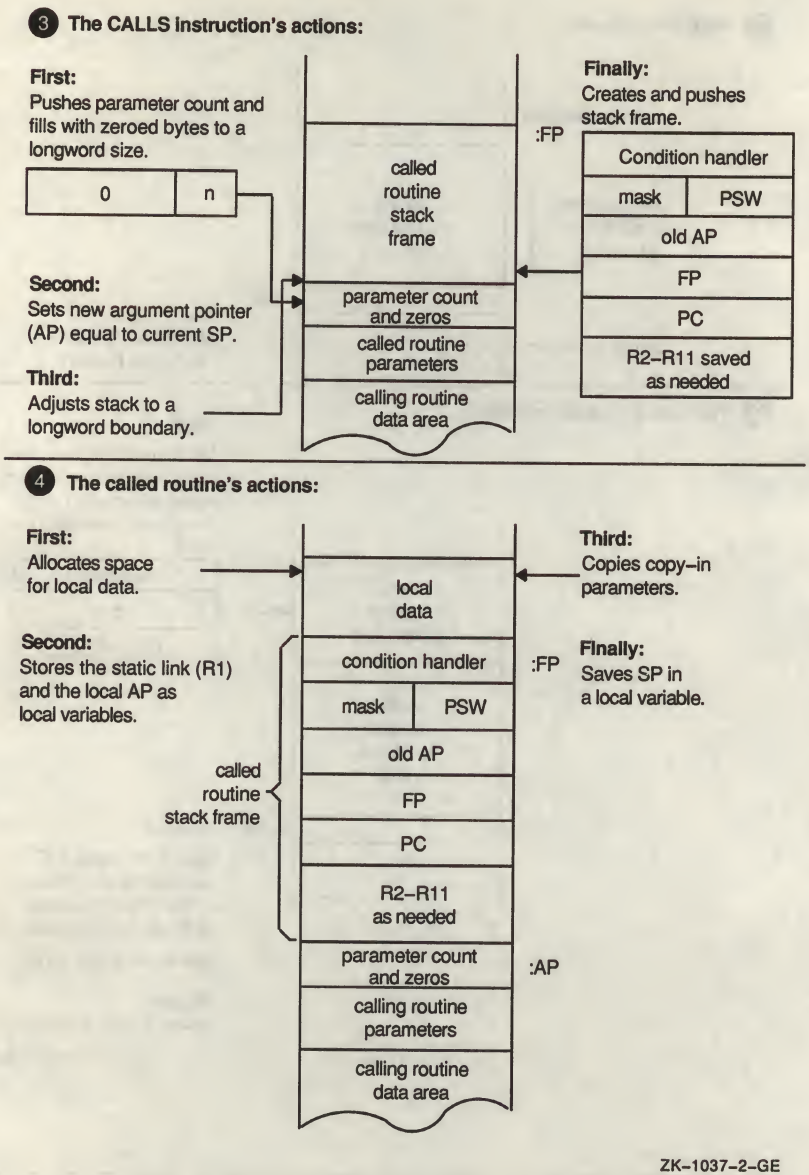
The VAX Pascal compiler uses the VAX CALLS and CALLG instructions to call routines. Figure 3–1 illustrates the events that occur during a routine call and shows the structure of the call stack after each event.

Figure 3–1: Contents of the Run-Time Stack



(continued on next page)

Figure 3–1 (Cont.): Contents of the Run-Time Stack



As shown in Figure 3–1, the frame pointer of the calling routine is stored in R1, thus creating a link between the calling routine and the called routine. Because this link exists, the called routine can access variables and routines declared in enclosing blocks and can use GOTO statements to access executable statements in enclosing blocks.

However, if you declare a routine with the UNBOUND attribute, the system does not assume that the frame pointer of the declaring routine is stored in R1; thus, no link is established. As a result, an unbound routine has the following restrictions:

- It cannot access automatic variables declared in enclosing blocks.
- It cannot call bound routines declared in enclosing blocks.
- It cannot use a GOTO statement to transfer control to enclosing blocks other than the main program block.

By default, routines declared at program or module level and all other routines declared with the INITIALIZE, GLOBAL, or EXTERNAL attributes have the characteristics of unbound routines. Routines passed by the immediate value mechanism must be UNBOUND.

Asynchronous system trap routines (ASTs) and RMS completion routines must have both the ASYNCHRONOUS and UNBOUND attributes. Because they are asynchronous, such routines can access only volatile variables, predeclared routines, and other asynchronous routines. Note that the VAX Pascal run-time system does not permit a program and an asynchronous routine (such as an AST) to access the same file simultaneously.

For More Information:

- On attributes (*VAX Pascal Reference Manual*)
- On the immediate value mechanism (Section 3.3.1)

3.2 Parameter-Passing Semantics

Parameter-passing semantics describe how parameters behave when passed between the calling and called routine. VAX Pascal passes parameter values by the following methods:

- Value passing semantics (Standard)
- Variable passing semantics (Standard)
- Foreign passing semantics (VAX Pascal extension)

By default, VAX Pascal passes arguments using value semantics.

For More Information:

For information on value, variable, and foreign semantics, see the *VAX Pascal Reference Manual.*

3.3 Parameter-Passing Mechanisms

The way in which an argument specifies how the actual data to be passed by the called routine is defined by the **parameter-passing mechanism**. In compliance with the VAX Procedure Calling Standard, VAX Pascal supports the following basic passing mechanisms:

- By immediate value—the longword argument in the argument list contains the actual data.
- By reference—the longword argument in the argument list contains the address of the data to be used by the routine.
- By descriptor—the longword argument in the argument list contains the address of a descriptor that describes the location, length, and data type of the data to be used by the routine.

By default, VAX Pascal uses the by reference mechanism to pass all actual parameters except those that correspond to conformant parameters and undiscriminated schema parameters, in which case the by descriptor mechanism is used. Table 3–2 describes the method you use in VAX Pascal to obtain the desired parameter-passing mechanism.

Table 3–2: Parameter-Passing Mechanisms

Parameter-Passing Mechanism	Methods Used in VAX Pascal
By immediate value	%IMMED or [IMMEDIATE]
By reference	Default or %REF
By descriptor	%DESCR, %STDESCR, [CLASS_S], [CLASS_A], or [CLASS_NCA]

A mechanism specifier usually appears before the name of a formal parameter, or if a passing attribute is used it appears in the attribute list of the formal parameter. However, in VAX Pascal, a mechanism specifier can also appear before the name of an actual parameter. In the latter case, the specifier overrides the type, passing semantics, passing mechanism, and the number of formal parameters specified in the formal parameter declaration.

For More Information:

For information on passing mechanisms and passing semantics, see Section 3.3.4.

3.3.1 By Immediate Value Passing Mechanism

The by immediate value passing mechanism passes a copy of a value instead of the address. VAX Pascal provides the %IMMED foreign passing mechanism and the IMMEDIATE attribute in order to pass a parameter by immediate value. You cannot use variable semantics with the by immediate value passing mechanism.

Because a by immediate value argument is only one longword in length, variables that require more than 32 bits of storage, including file variables, cannot be passed by immediate value.

3.3.2 By Reference Passing Mechanism

The by reference mechanism passes the address of the parameter to the called routine. This is the default parameter passing mechanism. When using the default by reference passing mechanism, the type of passing semantics used depends on the use of the VAR keyword. If the formal parameter name is preceded by the reserved word VAR, variable semantics is used; otherwise, value semantics is used.

In addition to using the defaults, the VAX Pascal compiler provides the %REF foreign passing mechanism and the REFERENCE attribute, which has more than one interpretation for the passing semantics depending on the data item represented by the actual parameter. This allows you the flexibility to have the called routine use either variable semantics or true foreign semantics. The mechanism specifier appears before the name of a formal parameter. The parameter passing attribute appears in the attribute list of the formal parameter.

3.3.3 By Descriptor Passing Mechanism

There are several types of descriptors. Each descriptor contains a value that identifies the descriptor's type. The called routine then uses the information held in the descriptor to identify its type and size.

When you use one of the VAX Pascal by descriptor mechanisms, the compiler passes the address of a string, array, or scalar descriptor. The VAX Pascal compiler generates the descriptor supplying the necessary information.

VAX Pascal provides three attributes for the by descriptor passing mechanism: [CLASS_S], [CLASS_A], and [CLASS_NCA]. With these three attributes, the type of passing semantics used for the by descriptor argument depends on the use of the VAR keyword. If the formal parameter name is preceded by the reserved word VAR, variable semantics is used; otherwise, value semantics is used. The parameter-passing attribute appears in the attribute list of the formal parameters.

Sometimes you may want the flexibility of choosing the passing semantics, either variable semantics or true foreign semantics. In this instance, the VAX Pascal compiler provides two foreign passing mechanism specifiers, %DESCR and %STDESCR. These specifiers have more than one interpretation for the passing semantics depending on the data type of the actual parameter. The mechanism specifier appears before the name of a formal parameter.

Table 3–3 lists the class and type of descriptor generated for parameters that can be passed using the by descriptor mechanism.

Table 3–3: Parameter Descriptors

Parameter Type	Descriptor Class and Type.		
	%DESCR	%STDESCR	Value or VAR Semantics
Ordinal	DSC$K_CLASS_S[1]	—	—
SINGLE	DSC$K_CLASS_S DSC$K_DTYPE_F	—	—
DOUBLE	DSC$K_CLASS_S DSC$K_DTYPE_D or DSC$K_DTYPE_G	—	—
QUADRUPLE	DSC$K_CLASS_S DSC$K_DTYPE_H	—	—
RECORD	—	—	—

[1]Descriptor's D_type depends on size of type.

(continued on next page)

Table 3–3 (Cont.): Parameter Descriptors

Parameter Type	Descriptor Class and Type.		
	%DESCR	%STDESCR	Value or VAR Semantics
ARRAY	DSC$K_CLASS_A[2]	DSC$K_CLASS_S DSC$K_DTYPE_T[3]	—
ARRAY OF VARYING OF CHAR	DSC$K_CLASS_VSA DSC$K_DTYPE_VT	—	—
Conformant ARRAY	DSC$K_CLASS_A[2]	DSC$K_CLASS_S DSC$K_DTYPE_T[3]	DSC$K_CLASS_A[2]
Conformant ARRAY OF VARYING OF CHAR[4]	DSC$K_CLASS_VSA DSC$K_DTYPE_VT	—	DSC$K_CLASS_VSA DSC$K_DTYPE_VT
VARYING OF CHAR	DSC$K_CLASS_VS DSC$K_DTYPE_VT	—	—
Conformant VARYING OF CHAR	DSC$K_CLASS_VS DSC$K_DTYPE_VT	—	DSC$K_CLASS_VS DSC$K_DTYPE_VT
STRING	—	—	DSC$K_CLASS_VS DSC$K_DTYPE_VS
Schema name	—	—	Internal VAX Pascal descriptor
Discriminated schema	—	—	—
SET	DSC$K_CLASS_S DSC$K_DTYPE_Z	—	—
FILE	DSC$K_CLASS_S DSC$K_DTYPE_Z	—	
Pointer	DSC$K_CLASS_S DSC$K_DTYPE_LU	—	—

[2]Descriptor's D_type depends on component type.
[3]Only if PACKED ARRAY OF CHAR.
[4]Component type can be a conformant VARYING OF CHAR.

(continued on next page)

Table 3–3 (Cont.): Parameter Descriptors

Parameter Type	Descriptor Class and Type.		
	%DESCR	%STDESCR	Value or VAR Semantics
PROCEDURE or FUNCTION	DSC$K_CLASS_S DSC$K_DTYPE_BPV	—	Bound procedure value by reference
	CLASS_A	**CLASS_NCA**	**CLASS_S**
Ordinal	—	—	DSC$K_CLASS_S[1]
SINGLE	—	—	DSC$K_CLASS_S DSC$K_DTYPE_F
DOUBLE	—	—	DSC$CLASS_S DSC$DTYPE_D or DSC$DTYPE_G
QUADRUPLE	—	—	DSC$CLASS_S DSC$DTYPE_H
RECORD	—	—	—
ARRAY	DSC$K_CLASS_A[2]	DSC$K_CLASS_NCA[2]	DSC$K_CLASS_S DSC$K_DTYPE_T[3]
ARRAY OF VARYING OF CHAR	—	—	—
Conformant ARRAY	DSC$K_CLASS_A[2]	DSC$K_CLASS_NCA[2]	DSC$K_CLASS_S DSC$K_DTYPE_T[3]
Conformant ARRAY OF VARYING OF CHAR[4]	—	—	—
VARYING OF CHAR	—	—	—

[1]Descriptor's D_type depends on size of type.
[2]Descriptor's D_type depends on component type.
[3]Only if PACKED ARRAY OF CHAR.
[4]Component type can be a conformant VARYING OF CHAR.

(continued on next page)

Table 3–3 (Cont.): Parameter Descriptors

Parameter Type	Descriptor Class and Type.		
	CLASS_A	CLASS_NCA	CLASS_S
Conformant VARYING OF CHAR	—	—	—
STRING	—	—	—
Schema name	—	—	—
Discriminated schema	—	—	—
SET	—	—	DSC$K_CLASS_S DSC$K_DTYPE_Z
FILE	—	—	DSC$K_CLASS_S DSC$K_DTYPE_Z
Pointer	—	—	DSC$K_CLASS_S DSC$K_DTYPE_LU
PROCEDURE or FUNCTION	—	—	—

3.3.3.1 CLASS_S Attribute

When the CLASS_S attribute is used on a formal parameter, the compiler generates a fixed-length scalar descriptor of a variable and passes its address to the called routine. Only ordinal, real, set, pointer, and one-dimensional packed arrays (fixed or conformant) that are of type OF CHAR can have the CLASS_S attribute on the formal parameter.

With the CLASS_S attribute, the type of passing semantics used for the by descriptor argument depends on the use of the VAR keyword. If the formal parameter name is preceded by the reserved word VAR, variable semantics is used; otherwise, value semantics is used.

3.3.3.2 CLASS_A and CLASS_NCA Attributes

When you use the CLASS_A or CLASS_NCA attribute on a formal parameter, the compiler generates an array descriptor and passes its address to the called routine. The type of the CLASS_A and CLASS_NCA parameter must be an array (packed or unpacked, fixed or conformant) of an ordinal or real type.

With the CLASS_A and CLASS_NCA attributes, the type of passing semantics used for the by descriptor argument depends on the use of the VAR keyword. If the formal parameter name is preceded by the reserved word VAR, variable semantics is used; otherwise, value semantics is used.

3.3.3.3 %STDESCR Mechanism Specifier

When you use the %STDESCR mechanism specifier, the compiler generates a fixed-length descriptor of a character-string variable and passes its address to the called routine. Only items of the following types can have the %STDESCR specifier on the actual parameter: character-string constants, string expressions, packed character arrays with lower bounds of 1, and packed conformant arrays with indexes of an integer or integer subrange type. The passing semantics depend on the variable represented by the actual parameter as follows:

- If the actual parameter is a variable of type PACKED ARRAY OF CHAR, %STDESCR implies variable semantics within the called routine.
- If the actual parameter is either a variable enclosed in parentheses, an expression, or a VARYING OF CHAR variable, %STDESCR implies foreign semantics.

If the actual parameter is not modified by the called external routine, the corresponding formal parameter should be declared READONLY, saving the copy from having to be made.

The following function declaration requires one fixed-length string descriptor as a parameter:

```
[ASYNCHRONOUS,EXTERNAL(SYS$SETDDIR)] FUNCTION $SETDDIR
   (%STDESCR New_Dir : PACKED ARRAY [$L1..$U1: INTEGER] OF CHAR;
   VAR Old_Dir_Len : $UWORD := %IMMED 0;
   VAR Old_Dir : [CLASS_S]PACKED ARRAY [$L2..$U2 : INTEGER] OF CHAR
                 := %IMMED 0) : INTEGER; EXTERN;

.
.
.
Status :=  $SETDDIR('[VAX_Pascal]');
```

The actual parameter '[VAX_Pascal]' is passed by string descriptor with foreign semantics to the formal parameter New_Dir.

3.3.3.4 %DESCR Mechanism Specifier

When you use the %DESCR mechanism specifier, the parameter generates a descriptor for an ordinal, real, or array variable and passes its address to the called routine. The type of %DESCR parameter can be any ordinal or real type, a VARYING OF CHAR string, or an array (packed or unpacked, fixed or conformant) of an ordinal or real type.

The passing semantics depend on the actual parameter's data type as follows:

- If the actual parameter is a variable, the %DESCR formal parameter implies variable semantics within the called routine.
- If the actual parameter is an expression or a variable enclosed in parentheses, %DESCR implies foreign semantics.

If the actual parameter is not modified by the called external routine, the corresponding formal parameter should be declared READONLY, saving the copy from having to be made.

The following function declaration requires a varying-length string descriptor as its parameter:

```
TYPE
   Vary = VARYING [30] OF CHAR;

VAR
   Obj_String  : Vary;
   Times_Found : INTEGER;

[EXTERNAL] FUNCTION Search_String( %DESCR String_Val : Vary )
   : BOOLEAN; EXTERNAL;

.
.
.
IF Search_String( Obj_String )
   THEN
   Times_Found := Times_Found + 1;
```

The actual parameter Obj_String is passed by varying string descriptor with variable semantics to the formal parameter String_Val.

For More Information:

For information on descriptor classes and types, see the *Introduction to VMS System Routines*.

3.3.4 Summary of Passing Mechanisms and Passing Semantics

Table 3–4 summarizes the passing semantics used when various mechanisms are specified on either the formal or the actual parameter. For example, if a variable is passed to a formal parameter that was declared without the keyword VAR and either %REF or [REFERENCE] was specified, then variable passing semantics will be used. Likewise, if a variable is passed to a formal parameter which was declared with the keyword VAR and either %REF or [REFERENCE] was specified, then an error will occur.

Note that if an actual parameter is preceded by a %IMMED specifier, regardless of what passing mechanism is used to declare the formal parameter, foreign semantics would be used, because a specifier appearing on the actual parameter always overrides the semantics specified on the formal parameter.

Table 3–4: Summary of Passing Mechanisms and Passing Semantics

	Actual Parameter			
	Variable		(Variable) or Expression	
Passing Mechanism	**No VAR on Formal**	**VAR on Formal**	**No VAR on Formal**	**VAR on Formal**
By immediate value				
%IMMED or [IMMEDIATE]	Foreign	Error	Foreign	Error
By reference				
default	Value	Variable	Value	Value[1]
%REF or [REFERENCE]	Variable	Error	Foreign	Error
By descriptor				
[CLASS_S]	Value	Variable	Value	Value[1]
[CLASS_A]	Value	Variable	Value	Value[1]
[CLASS_NCA]	Value	Variable	Value	Value[1]
%STDESCR	Variable	Error	Foreign	Error
%DESCR	Variable	Error	Foreign	Error

[1]If the formal parameter is declared with the READONLY attribute, then value passing semantics is used; otherwise, it is an error.

3.4 Passing Parameters from VAX Pascal Routines to Non-VAX Pascal Routines

Non-VAX Pascal routines require parameters in the form of addresses, immediate values, or descriptors. As described in the previous sections, VAX Pascal provides various ways to pass parameters to the called routine. You can specify the desired passing mechanism by using one of the mechanism specifiers %REF, %IMMED, %DESCR, or %STDESCR, or by specifying one of the passing mechanism attributes [REFERENCE], [IMMEDIATE], [CLASS_S], [CLASS_A], or [CLASS_NCA].

You can also specify the by reference mechanism by indicating value and variable semantics in the declarations of formal parameters to non-Pascal routines. Table 3–5 illustrates the types of parameters that can be passed to foreign mechanism parameters.

Table 3–5: Foreign Mechanism Parameters

	Passing Mechanism						
Parameter Type	**%IMMED**	**%REF**	**%DESCR**	**%STDESCR**	**CLASS_S**	**CLASS_A**	**CLASS_NCA**
Ordinal	Yes	Yes	Yes	No	Yes	No	No
SINGLE	Yes	Yes	Yes	No	Yes	No	No
DOUBLE or QUADRUPLE	No	Yes	Yes	No	Yes	No	No
RECORD	Yes[1]	Yes	No	No	No	No	No
ARRAY	Yes[1]	Yes	Yes	Yes[2]	Yes[2]	Yes	Yes
ARRAY OF VARYING OF CHAR	No	Yes	Yes	No	No	No	No
Conformant ARRAY	Yes[1]	Yes	Yes	Yes[2]	Yes[2]	Yes	Yes

[1]Allocation size must be less than or equal to 32 bits.
[2]Only if PACKED ARRAY OF CHAR.

(continued on next page)

Table 3–5 (Cont.): Foreign Mechanism Parameters

Parameter Type	Passing Mechanism						
	%IMMED	%REF	%DESCR	%STDESCR	CLASS_S	CLASS_A	CLASS_NCA
Conformant ARRAY OF VARYING OF CHAR[3]	No	Yes	Yes	No	No	No	No
VARYING OF CHAR	No	Yes	Yes	No	No	No	No
Conformant VARYING OF CHAR	No	Yes	Yes	No	No	No	No
STRING	No	No	No	No	No	No	No
Schema name	No	No	No	No	No	No	No
Discriminated schema	No	No	No	No	No	No	No
SET	Yes[1]	Yes	Yes	No	Yes	No	No
FILE	No	Yes	No	No	No	No	No
Pointer	Yes	Yes	Yes	No	Yes	No	No
PROCEDURE or FUNCTION	Yes[4]	Yes	Yes	No	No	No	No

[1]Allocation size must be less than or equal to 32 bits.
[3]Component type can be a conformant VARYING OF CHAR.
[4]Must be unbound.

3.5 Passing Parameters from Non-VAX Pascal Routines to VAX Pascal Routines

When calling a VAX Pascal routine from a non-VAX Pascal routine, you must ensure that the parameters are in the form required by the VAX Pascal routine. By default, VAX Pascal requires most parameters to be passed by reference. The following list describes additional requirements:

- When the VAX Pascal routine requires a value parameter, the parameter list of the calling routine must contain the address of a value. The VAX Pascal routine will copy the value from the passed address upon entry.
- When the VAX Pascal routine requires a VAR parameter, the parameter list of the calling routine must contain the address of a variable. The VAX Pascal routine uses the address to access the actual parameter variable. An actual parameter variable whose value can change as a result of routine execution must be passed in this manner. In addition, all files must be passed to VAX Pascal routines as VAR parameters.
- When the VAX Pascal routine requires a formal procedure or function parameter, the parameter list of the calling routine must specify the address of a bound procedure value. This process implements the VAX by reference mechanism for a routine.
- When the VAX Pascal routine requires a formal conformant array, the parameter list of the calling routine must contain the address of a CLASS_A descriptor.
- When the VAX Pascal routine requires a formal conformant VARYING parameter, the parameter list of the calling routine must contain the address of a CLASS_VS descriptor.
- When the VAX Pascal routine requires a formal undiscriminated STRING parameter, the parameter list of the calling routine must contain the address of a varying string descriptor.
- When the VAX Pascal routine requires a discriminated or undiscriminated schema parameter, the mechanism used is private to the VAX Pascal language and is not available to other languages.

Chapter 4

VMS System Routines

To eliminate duplication of programming and debugging efforts, the VMS system provides many routines to perform common programming tasks. These routines are collectively known as system routines. They include routines in the VMS Run-Time Library to assist you in such areas as mathematics, screen management, and string manipulation. Also included are VAX Record Management Services (RMS), which are used to access files and their records. There are also system services that perform tasks such as resource allocation, information sharing, and input/output coordination.

This chapter discusses the following topics:

- System definitions files (Section 4.1).
- Declaring system routines (Section 4.2)
- Calling system routines (Section 4.3)

Because all VMS system routines adhere to the VAX Procedure Calling Standard, you can declare any system routine as an external routine and then call the routine from a VAX Pascal program.

4.1 System Definitions Files

To access system entry points, data structures, symbol definitions, and messages, VAX Pascal provides files that you can inherit (.PEN) or include (.PAS) in your application. Table 4–1 summarizes the source and environment files that VAX Pascal makes available to you in the directory SYS$LIBRARY (for instance, SYS$LIBRARY:STARLET.PEN).

Table 4–1: VAX Pascal Definitions Files

File	Description
System Services Definitions File:	
STARLET.PAS STARLET.PEN	Contains VMS system service definitions, LIB$ messages, MTH$ messages, OTS$ messages, SMG$ data structures and "termtable," STR$ messages, RMS routine declarations, system symbolic names, and RMS data structures.
Individual Definitions Files:[1]	
PASCAL$DTK_ROUTINES.PAS PASCAL$DTK_ROUTINES.PEN	Contains DTK$ routine entry points, data structures, and messages.
PASCAL$LIB_ROUTINES.PAS PASCAL$LIB_ROUTINES.PEN	Contains LIB$ routine entry points.
PASCAL$MTH_ROUTINES.PAS PASCAL$MTH_ROUTINES.PEN	Contains MTH$ routine entry points.
PASCAL$NCS_ROUTINES.PAS PASCAL$NCS_ROUTINES.PEN	Contains NCS$ routine entry points.
PASCAL$OTS_ROUTINES.PAS PASCAL$OTS_ROUTINES.PEN	Contains OTS$ routine entry points.
PASCAL$PPL_ROUTINES.PAS PASCAL$PPL_ROUTINES.PEN	Contains PPL$ routine entry points, data structures, and messages.
PASCAL$SMG_ROUTINES.PAS PASCAL$SMG_ROUTINES.PEN	Contain SMG$ routine entry points and messages.
PASCAL$SOR_ROUTINES.PAS PASCAL$SOR_ROUTINES.PEN	Contains SOR$ routine entry points and messages.
PASCAL$STR_ROUTINES.PAS PASCAL$STR_ROUTINES.PEN	Contains STR$ routine entry points.

[1]VAX Pascal added these libraries as of VMS Version 5.0.

(continued on next page)

Table 4–1 (Cont.): VAX Pascal Definitions Files

File	Description
Symbol Definitions Files:[2]	
LIBDEF.PAS	Contains definitions for all condition symbols from the general utility VMS Run-Time Library routines.
MTHDEF.PAS	Contains definitions for all condition symbols from the mathematical routines library.
SIGDEF.PAS	Contains miscellaneous symbol definitions used in condition handlers. These definitions are also included in STARLET.PEN.
VAX Pascal Run-Time Library Files:	
PASDEF.PAS	Contains definitions for all condition symbols from the VAX Pascal Run-Time Library routines.
PASSTATUS.PAS	Contains definitions for all values returned by the STATUS and STATUSV routines.

[2]These files are retained for compatiblity with VAX Pascal Version 1.0 and do not contain symbol definitions for subsequent releases of the product. (For definitions that are complete for the latest release of VMS, use the individual PASCAL$ files or STARLET. To access these files, use the %INCLUDE directive in the CONST declaration section of your program.

For instance, the external routine declarations in STARLET define new identifiers by which you can refer to the routines. Example 4–1 shows that you can refer to SYS$HIBER as $HIBER if you use STARLET.

Example 4–1: Inheriting STARLET.PEN to call SYS$HIBER

```
[INHERIT('SYS$LIBRARY:STARLET')] PROGRAM Suspend (INPUT,OUTPUT);

TYPE
   Sys_Time = RECORD
      I,J : INTEGER;
      END;
   Unsigned_Word = [WORD] 0..65535;

VAR
   Current_Time : PACKED ARRAY[1..80] OF CHAR;
   Length       : Unsigned_Word;
   Job_Name     : VARYING[15] OF CHAR;
   Ascii_Time   : VARYING[80] OF CHAR;
   Binary_Time  : Sys_Time;

BEGIN
{ Print current date and time }
$ASCTIM (TIMLEN := Length, TIMBUF := Current_Time);
WRITELN ('The current time is ', SUBSTR(Current_Time, 1, Length);

{ Get name of process to suspend }
WRITE ('Enter name of process to suspend: ');
READLN (Job_Name);

{ Get time to wake process }
WRITE ('Enter time to wake process: ');
READLN (Ascii_Time);

{ Convert time to binary }
IF NOT ODD ($BINTIM (Ascii_Time, Binary_Time))
THEN
   BEGIN
      WRITELN ('Illegal format for time string');
      HALT;
   END;

{ Suspend process }
IF NOT ODD ($SUSPND (PRCNAM := Job_Name))
THEN
   BEGIN
      WRITELN ('Cannot suspend process');
      HALT;
   END;

{ Schedule wakeup request for self }
IF ODD ($SCHDWK (DAYTIME := Binary_Time))
THEN
   $HIBER { Put self to sleep }
ELSE
   BEGIN
      WRITELN ('Cannot schedule wakeup');
      WRITELN ('Process will resume immediately');
   END;
```

(continued on next page)

Example 4–1 (Cont.): Inheriting STARLET.PEN to call SYS$HIBER

```
{ Resume process }
IF NOT ODD ($RESUME (PRCNAM := Job_Name))
THEN
   BEGIN
      WRITELN ('Cannot resume process');
      HALT;
   END;
END.
```

4.2 Declaring System Routines

Before calling a routine, you must declare it. System routine names conform to one of the two following conventions:

```
[[facility-code]]$procedure-name
```

For example, LIB$PUT_OUTPUT is the Run-Time Library routine used to write a record to the current output device and $ASCTIM is a system service routine used to convert binary time to ASCII time.

Because system routines are often called from condition handlers or asynchronous trap (AST) routines, you should declare system routines with the ASYNCHRONOUS attribute.

Each system routine is documented with a structured format in the appropriate VMS reference manual. The documentation for each routine describes the routine's purpose, the declaration format, the return value, and any required or optional parameters. Detailed information about each parameter is listed in the description. The following format is used to describe each parameter.

```
parameter-name
VMS Usage :     VMS data type
type      :     parameter data type
access    :     parameter access
mechanism :     parameter-passing mechanism
```

Using this information you must determine the parameter's data type (type), the parameter's passing semantics (access), the mechanism used to pass the parameter (mechanism), and whether the parameter is required or optional from the call format.

The following sections describe the methods available in VAX Pascal to obtain the various data types, access methods, and passing mechanisms.

4.2.1 Methods Used to Obtain VMS Data Types

The data specified by a parameter has a data type. Several VMS standard data types exist. A system routine parameter must use one of these data types.

For More Information:

For information on VMS data types and equivalent VAX Pascal declarations, see the *Introduction to VMS System Routines*.

4.2.2 Methods Used to Obtain Access Methods

The access method describes the way in which the called routine accesses the data specified by the parameter. The following three methods of access are the most common:

- Read only—data must be read by the called routine. The called routine does not write the input data. Thus, input data supplied by a variable is preserved when the called routine completes execution.
- Write only—data that the called routine returns to the calling routine must be written into a location where the calling routine can access it. Such data can be thought of as output data. The called routine does not read the contents either before or after it writes into the location.
- Modify—a parameter specifies data that is both read and written by the called routine. In this case, the called routine reads the input data, which it uses in its operations, and then overwrites the input data with the results. Thus, when the called routine completes execution, the input data specified by the argument is lost.

Table 4–2 lists all access methods that may appear under the access entry in a parameter description, as well as the VAX Pascal translation.

Table 4–2: Access Type Translations

Access Entry	Method Used in VAX Pascal
Call after stack unwind	Procedure or function parameter passed by immediate value
Function call (before return)	Function parameter
Jump after unwind	Not available
Modify	Variable semantics[1]
Read only	Value or foreign semantics
Call without stack unwind	Procedure parameter
Write only	Variable semantics[1]

[1]It is possible to obtain variable semantics by either specifying the VAR keyword on the formal parameter or by passing a variable as an actual parameter using %REF, %DESCR, or %STDESCR.

4.2.3 Methods Used to Obtain Passing Mechanisms

The way in which an argument specifies the actual data to be used by the called routine is defined in terms of the parameter-passing mechanism.

Table 4–3 lists all passing mechanisms that may appear under the mechanism entry in an argument description and the method used in VAX Pascal to obtain the passing mechanism.

Table 4–3: Mechanism Type Translations

Mechanism Entry	Method Used in VAX Pascal
By value	%IMMED or [IMMEDIATE]
By reference	VAR, %REF or [REFERENCE] or default
By descriptor	
Fixed-length	%STDESCR parameter of type PACKED ARRAY OF CHAR or [CLASS_S]
Dynamic-string	%STDESCR parameter of type PACKED ARRAY OF CHAR or [CLASS_S]

(continued on next page)

Table 4–3 (Cont.): Mechanism Type Translations

Mechanism Entry	Method Used in VAX Pascal
Array	Array type, conformant array parameter, or [CLASS_A]
Procedure	n.a.
Decimal-string	n.a.
Noncontiguous-array	%DESCR or [CLASS_NCA]
Varying-string	Value, VAR or %DESCR conformant parameter of type VARYING OF CHAR, or %DESCR parameter of type VARYING OF CHAR
Varying-string-array	Value, VAR or %DESCR conformant parameter of type array of VARYING OF CHAR, or %DESCR parameter of type array of VARYING OF CHAR
Unaligned-bit-string	n.a.
Unaligned-bit-array	n.a.
String-with-bounds	n.a.
Unaligned-bit-string-with-bounds	n.a.

Parameters passed by reference and used solely as input to a system service should be declared with VAX Pascal value semantics; this allows actual parameters to be compile-time and run-time expressions. When a system service requires a formal parameter with a mechanism specifier, you should declare the formal parameter with the READONLY attribute to specify value semantics. Other parameters passed by reference should be declared with VAX Pascal variable semantics to ensure that the output data is interpreted correctly. In some cases, by reference parameters are used for both input and output and should also be declared with variable semantics.

The following example shows the declaration of the Convert ASCII String to Binary Time (SYS$BINTIM) system service and a corresponding function designator:

```
TYPE
   $QUAD = [QUAD,UNSAFE] RECORD
              L0 : UNSIGNED;
              L1 : INTEGER;
           END;

VAR
   Ascii_Time  : VARYING[80] OF CHAR;
   Binary_Time : $QUAD;

[ASYNCHRONOUS,EXTERNAL(SYS$BINTIM)] FUNCTION $BINTIM
   (TIMBUF : [CLASS_S] PACKED ARRAY [$l1..$u1:INTEGER] OF CHAR;
   VAR TIMADR : [VOLATILE] $QUAD)
   : INTEGER; EXTERNAL;
{In the executable section:}
IF NOT ODD ($BINTIM(Ascii_Time, Binary_Time))
THEN
   BEGIN
      WRITELN ('Illegal format for time string');
      HALT;
   END;
```

The first formal parameter requires the address of a character-string descriptor with value semantics; the second requires an address and uses variable semantics to manipulate the parameter within the service. Because you can call $BINTIM from a condition handler or AST routine, you should declare it with the ASYNCHRONOUS attribute. Also, because you may want to pass a volatile variable to the TIMADR parameter, you should use the VOLATILE attribute to indicate that the argument is allowed to be volatile.

4.2.4 Data Structure Parameters

Some system services require a parameter to be the address of a data structure that indicates a function to be performed or that holds information to be returned. Such a structure can be described as a list, a control block, or a vector. The size and POS attributes provide an efficient method of laying out these data structures. The size attributes ensure that the fields of the data structure are of the size required by the system service, and the POS attribute allows you to position the fields correctly.

For example, the Get Job/Process Information (SYS$GETJPIW) system service requires an item list consisting of an array of records of 12 bytes, where all but the last array cell requests one piece of data and the last array cell represents the item list terminator. By packing the record, you can guarantee that the fields of each record are allocated contiguously.

Example 4–2 uses the system service routine $GETJPIW to retrieve the process's name as a 12-byte string.

Example 4–2: Using $GETJPIW to Retrieve a Process Name

```
[INHERIT('SYS$LIBRARY:STARLET')] PROGRAM Userid( OUTPUT );

TYPE
   Uword       = [WORD] 0..65535;
   Itmlst_Cell = PACKED RECORD
                    CASE INTEGER OF
                       1 : (Buf_Len   : Uword;
                            Item_Code : Uword;
                            Buf_Addr  : INTEGER;
                            Len_Addr  : INTEGER);
                       2 : (Term      : INTEGER);
                 END;
VAR
   Username_String   : [VOLATILE] VARYING [12] OF CHAR;
   Itmlst            : ARRAY [1..2] OF Itmlst_Cell := ZERO;
BEGIN
Itmlst[1].Buf_Len := 12;                    { 12 bytes returned }
Itmlst[1].Item_Code := JPI$_USERNAME;       { return user name }
Itmlst[1].Buf_Addr :=                       { store returned name here }
   IADDRESS(Username_String.BODY);
Itmlst[1].Len_Addr :=                       { store returned length here }
   IADDRESS(Username_String.LENGTH);
Itmlst[2].Term := 0;                        { terminate item list }

IF NOT ODD( $GETJPIW(,,,Itmlst) )
THEN
   WRITELN('error')
ELSE
   WRITELN('user name is ',Username_String);
END.
```

For More Information:

For information on size attributes, see the *VAX Pascal Reference Manual.*

4.2.5 Default Parameters

In some cases, you do not have to supply actual parameters to correspond to all the formal parameters of a system service. In VAX Pascal, you can supply default values for such optional parameters when you declare the service. You can then omit the corresponding actual parameters from the routine call. If you choose not to supply an optional parameter, you should initialize the formal parameter with the appropriate value, using the by immediate value (%IMMED) mechanism. Usually, the correct default value is 0.

For example, the Cancel Timer (SYS$CANTIM) system service has two optional parameters. If you do not specify values for them in the actual parameter list, you must initialize them with zeros when they are declared. The following example is the routine declaration for SYS$CANTIM:

```
[ASYNCHRONOUS,EXTERNAL(SYS$CANTIM)] FUNCTION $CANTIM (
        %IMMED REQIDT : UNSIGNED := %IMMED 0;
        %IMMED ACMODE : UNSIGNED := %IMMED 0) : INTEGER; EXTERNAL;
```

A call to $CANTIM must indicate the position of omitted parameters with a comma, unless they all occur at the end of the parameter list. For example, the following are legal calls to $CANTIM using the above external declaration:

```
$CANTIM (, PSL$C_USER);
$CANTIM (I);
$CANTIM;
```

PSL$C_USER is a symbolic constant that represents the value of a user access mode, and I is an integer that identifies the timer request being canceled. Note that if you call $CANTIM with both of its default parameters, you can omit the actual parameter list completely.

When it is possible for the parameter list to be truncated, you can also specify the TRUNCATE attribute on the formal parameter declaration of the optional parameter. The TRUNCATE attribute indicates that an actual parameter list for a routine can be truncated at the point that the attribute was specified. However, once one optional parameter is omitted in the actual parameter list, it is not possible to specify any optional parameter following that. For example:

```
[ASYNCHRONOUS] FUNCTION LIB$GET_FOREIGN (
   VAR Resultant_String : [CLASS_S,VOLATILE]
           PACKED ARRAY [$l1..$u1:INTEGER] OF CHAR;
        Prompt_String   : [CLASS_S,TRUNCATE]
           PACKED ARRAY [$l2..$u2:INTEGER] OF CHAR := %IMMED 0;
   VAR Resultant_Length : [VOLATILE,TRUNCATE] $UWORD := %IMMED 0;
   VAR Flags            : [VOLATILE,TRUNCATE] UNSIGNED := %IMMED 0)
           : INTEGER; EXTERNAL;
```

With this declaration, it is possible to specify values for Resultant_String and Prompt_String and truncate the call list at that point. In this case, two parameters would be passed in the CALL instruction.

You may want to use a combination of the %IMMED 0 and TRUNCATE methods. This combination would allow you to skip the specification of intermediate optional parameters, as well as allow you to truncate the call list once all desired parameters have been specified.

Note that VMS system services require a value to be passed by parameters, including optional parameters; therefore, you should not use the TRUNCATE attribute when defining optional parameters to a system service. Instead, you should specify default values on the formal parameter declaration.

The TRUNCATE attribute is useful when calling routines for which the optional parameter is truly optional, for example, when calling VMS Run-Time Library routines.

For More Information:

For information on the TRUNCATE attribute, see the *VAX Pascal Reference Manual*.

4.2.6 Arbitrary Length Parameter Lists

Some Run-Time Library routines require a variable number of parameters. For example, there is no fixed limit on the number of values that can be passed to functions that return the minimum or maximum value from a list of input parameters. The LIST attribute supplied by VAX Pascal allows you to indicate the mechanism by which excess actual parameters are to be passed. For example:

```
[ASYNCHRONOUS] FUNCTION MTH$DMIN1 (
   D_FLOATING    : DOUBLE;
   EXTRA_PARAMS : [LIST] DOUBLE) : DOUBLE; EXTERNAL;
```

Because the function MTH$DMIN1 returns the D_floating minimum of an arbitrary number of D_floating parameters, the formal parameter EXTRA_PARAMS is declared with the LIST attribute. All actual parameters must be double-precision real numbers passed by reference with value semantics.

For More Information:

For information on the LIST attribute, see the *VAX Pascal Reference Manual*.

4.3 Calling System Routines

All system routines are functions that return an integer condition value; this value indicates whether the function executed successfully. An odd-numbered condition value indicates successful completion; an even-numbered condition value indicates a warning message or failure. Your program can use the VAX Pascal predeclared function ODD to test the function return value for success or failure. For example:

```
IF NOT ODD ($BINTIM(Ascii_Time,Binary_Time))
THEN
   BEGIN
      WRITELN('Illegal format for time string');
      HALT;
   END;
```

In addition, Run-Time Library routines return one or two values: the result of a computation or the routine's completion status, or both. When the routine returns a completion status, you should verify the return status before checking the result of a computation. You can use the function ODD to test for success or failure or you can check for a particular return status by comparing the return status to one of the status codes defined by the system. For example:

```
VAR
   Seed_Value  : INTEGER;
   Rand_Result : REAL;

[ASYNCHRONOUS] FUNCTION MTH$RANDOM (
   VAR seed : [VOLATILE] UNSIGNED) : SINGLE; EXTERNAL;
{In the executable section:}
Rand_Result := MTH$RANDOM (Seed_Value);
```

When the routine's completion status is irrelevant, your program can treat the function as though it were an external procedure and ignore the return value. For example, your program can declare the Hibernate (SYS$HIBER) system service as a function but call it as though it were a procedure:

```
[ASYNCHRONOUS,EXTERNAL(SYS$HIBER)] FUNCTION $HIBER
   : INTEGER; EXTERNAL;
{In the executable section:}
$HIBER; { Put process to sleep }
```

Because SYS$HIBER is expected to execute successfully, the program will ignore the integer condition value that is returned.

Chapter 5

Input and Output Processing

This chapter provides details on the I/O support provided by VMS on VAX machines and discusses the following topics:

- Environment I/O support (Section 5.1)
- User-action functions (Section 5.2)
- File sharing (Section 5.3)
- Record locking (Section 5.4)

5.1 Environment I/O Support

VAX Pascal uses the Record Management Services (RMS) to perform I/O tasks at the system level. In this environment, all of the VAX Pascal I/O model is supported; the model is based on RMS concepts. If these sections contain no information on a concept or element in the VAX Pascal I/O model, then this environment supports the concept or element exactly as it is described in the *VAX Pascal Reference Manual*.

You can use RMS features through Pascal when you call the OPEN procedure. For instance, when you call this procedure, you can specify the file organization, the component format, and the access method.

If you choose to use additional features of RMS that are not available in the VAX Pascal I/O model, you can write a user-action function that manipulates the RMS control blocks: the file-access block (FAB), the record access block (RAB), and the extended attribute block (XAB). Once you write the user-action function, you pass the function name as a parameter to the OPEN procedure.

For More Information:

- On user-action functions (Section 5.2)
- On OPEN defaults (Section 5.1.6.1)
- On OPEN and the VAX Pascal I/O model (*VAX Pascal Reference Manual*)
- On RMS concepts (*Guide to VMS File Applications*)
- On the user interface to RMS (*VAX Record Management Services Manual*)

5.1.1 Indexed Files

The VAX Pascal I/O model allows you to use most of the features of RMS indexed files. However, if you wish to use segmented or null keys, you must write a user-action function.

When an existing indexed file is opened, the Run-Time Library compares the keys in the file against the KEY attributes specified in the program. If no KEY attribute was specified for the corresponding key in the indexed file, then the comparison is bypassed and the open continues. The Run-Time Library compares the position and the data type of the file's keys against the KEY attributes specified. If the KEY attribute explicitly specifies a collating sequence (ASCENDING or DESCENDING), then the specified sequence must match that of the key in the file. If no sequence is specified, either sequence is allowed. The CHANGES and DUPLICATES options are not checked.

For More Information:

- On user-action functions (Section 5.2)
- On the OPEN procedure (Section 5.1.6)
- On indexed file organization and the KEY attribute (*VAX Pascal Reference Manual*)

5.1.2 "Components" and "Records"

In the VAX Pascal I/O model, data items in a file are called "components." In RMS, these items are called "records."

5.1.3 Count Fields for Variable-Length Components

Each variable-length component contains a count field as a prefix. This count field contains the number of bytes in the rest of the component. For files on tape, this count field is 4 bytes in length; for files on disk, this count field is 2 bytes in length.

5.1.4 Variable-Length with Fixed-Length Control Field (VFC) Component Format

The VAX Pascal I/O model does not provide a direct means to create files of variable-length components with fixed-length control fields (VFC). If you open a file of this component format, VAX Pascal treats the file like a file of variable-length components. If you want to create files of this component format, you must write a user-action function.

For More Information:

- On user-action functions (Section 5.2)
- On VFC components (*Guide to VMS File Applications*)

5.1.5 Random Access by Record File Address (RFA)

The VAX Pascal I/O model does not allow random access by record file address. If you want to use this type of access, you must write a user-action function.

RMS supports random access by Record File Address (RFA) for relative and indexed files, and for sequential files only on disk. The RFA is a unique number supplied for files on disk. The RFA remains constant as long as the record is in the file. RMS makes the RFA available to your program every time the record is stored or retrieved. Your program can either ignore the RFA or it can keep it as a random-access pointer to the record for subsequent accesses.

If your disk file is sequential with variable-length records, the RFA provides the only method for randomly accessing records of that file.

For More Information:

- On user-action functions (Section 5.2)
- On RFA (*Guide to VMS File Applications*)
- On using a record file address (*VAX Pascal User Manual*)

5.1.6 OPEN Procedure

When you use the OPEN procedure, RMS applies default values for VMS file specifications, and assigns values to FAB, RAB, XAB, and Name Block data structures.

5.1.6.1 OPEN Defaults

When you use OPEN to open a file, RMS applies certain defaults when attempting to locate the physical file. Table 5–1 presents these defaults.

Table 5–1: Default Values for VMS File Specifications

Element	Default
Node	Local computer
Device	Current user device
Directory	Current user directory
File name	VAX Pascal file variable name or its logical name translation
File type	.DAT
Version number (history)	OLD: highest current number NEW: highest current number + 1

The OPEN procedure includes a default file-name parameter. Using this parameter, you can access the RMS default file-name parameter to set file-specification defaults. Consider the following:

```
VAR
    My_File : VARYING [20] OF CHAR;
    My_File_Var : TEXT;
BEGIN
My_File := 'foo.bar';
OPEN( FILE_NAME := My_File,
      FILE_VARIABLE := My_File_Var,
      DEFAULT := '[another.dir]' );
```

The OPEN statement in the previous example opens the file called [ANOTHER.DIR]FOO.BAR. RMS applies the defaults in Table 5–1 to determine the node, the device, and the version number of the file.

For More Information:

- On file specifications (*Introduction to VMS*)
- On the OPEN procedure (*VAX Pascal Reference Manual*)

5.1.6.2 OPEN and RMS Data Structures

Table 5–2 presents the status of RMS FAB fields when you call the OPEN procedure. If a field is not included in the following tables, it is initialized to zero.

Table 5–2: Setting of RMS File Access Block Fields by Call to OPEN

Field	Name	OPEN Parameters and Value
FAB$L_CTX	Context	Reserved to Digital.
FAB$L_DEV	Device characteristics	Returned by RMS.
FAB$L_DNA	Default file specification string address	DEFAULT parameter value, if specified; else, ".DAT".
FAB$L_DNS	Default file specification string size	Set to length of default file name string.
FAB$B_FAC	File access options	
FAB$V_DEL	Allow deletions	1, if not HISTORY:=READONLY.
FAB$V_GET	Allow reads	1.
FAB$V_PUT	Allow writes	1, if not HISTORY:=READONLY.
FAB$V_TRN	Allow truncations	1, if not HISTORY:=READONLY.
FAB$V_UPD	Allow updates	1, if not HISTORY:=READONLY.
FAB$L_FNA	File specification string address	FILE_NAME if specified, name of file variable if external file, else 0.
FAB$B_FNS	File specification string size	Set to length of file name string.
FAB$L_FOP	File processing options	
FAB$V_CIF	Create if nonexistent	1, if HISTORY := UNKNOWN.

(continued on next page)

Table 5–2 (Cont.): Setting of RMS File Access Block Fields by Call to OPEN

Field	Name	OPEN Parameters and Value
FAB$V_DFW	Deferred write	1.
FAB$V_DLT	Delete on close service	Set when file is closed, depends on DISPOSITION.
FAB$V_NAM	Name block inputs	1, if terminal file reopened to enable prompting.
FAB$V_SFC	Submit command file (when closed)	Set when file is closed, depends on DISPOSITION.
FAB$V_SQO	Sequential only	1, if ACCESS_METHOD:=SEQUENTIAL (default).
FAB$V_TEF	Truncate at end of file	Initialized to 0, set to 1 after REWRITE or TRUNCATE of a sequential organization file.
FAB$V_TMD	Temporary (marked for deletion)	1, if nonexternal file with no FILE_NAME specified and DISPOSITION:=DELETE specified or implied.
FAB$B_FSZ	Fixed control area size	2, if terminal file enabled for prompting.
FAB$W_IFI	Internal file identifier	Returned by RMS.
FAB$W_MRS	Maximum record size	RECORD_LENGTH if specified; file component size if ORGANIZATION is not SEQUENTIAL or if RECORD_TYPE:=FIXED.
FAB$L_NAM[1]	Name block address	Set to address of name block (the expanded and resultant string areas are set up, but the related file name string is not).
FAB$B_ORG	File organization	FAB$C_REL if ORGANIZATION:=RELATIVE; FAB$C_IDX if ORGANIZATION:=INDEXED; FAB$C_SEQ in all other cases.
FAB$B_RAT	Record attributes	
FAB$V_FTN	FORTRAN carriage control	1, if CARRIAGE_CONTROL:=FORTRAN.
FAB$V_CR	Add LF and CR	1, if CARRIAGE_CONTROL:=LIST (default for TEXT and VARYING OF CHAR files).
FAB$V_PRN	Print file format	1, if terminal file enabled for prompting.

[1]After the call to OPEN, FAB$L_NAM must contain the same value it had before the call.

(continued on next page)

Table 5–2 (Cont.): Setting of RMS File Access Block Fields by Call to OPEN

Field	Name	OPEN Parameters and Value
FAB$B_RFM	Record format	FAB$C_FIX if RECORD_TYPE:=FIXED or if file component is of fixed size; FAB$C_VAR if RECORD_TYPE:=VARIABLE or file is VARYING or TEXT; FAB$C_STM if RECORD_TYPE:=STREAM; FAB$C_STMCR if RECORD_TYPE:=STREAM_CR; FAB$C_STMLF if RECORD_TYPE:=STREAM_LF; FAB$C_VFC if a terminal file enabled for prompting.
FAB$L_SDC	Spooling device characteristics	Returned by RMS.
FAB$L_XAB[2]	Extended attribute block address	The XAB chain always has a File Header Characteristics (FHC) extended attribute block in order to get longest record length (XAB$W_LRL). If ACCESS_METHOD:=KEYED, key index definition blocks are also present. Digital may add additional XABs in the future. Your user-action function may insert XABs anywhere in the chain. This field is only valid during execution of user-action functions; VAX Pascal places 0 in this field after the call to OPEN.
FAB$B_SHR	File sharing	
FAB$V_SHRPUT	Allow other PUTs	1, if SHARING:=READWRITE.
FAB$V_SHRGET	Allow other GETs	1, if SHARING is not NONE (default if HISTORY:=READONLY).
FAB$V_SHRDEL	Allow other DELETEs	1, if SHARING:=READWRITE.
FAB$V_SHRUPD	Allow other UPDATEs	1, if SHARING:=READWRITE.
FAB$V_NIL	Allow no other operations	1, if SHARING:=NONE (default if HISTORY is not READONLY).

[2]You cannot change XABs provided by Digital, but you can add and delete XABs that you insert using a user-action function.

Table 5–3 presents the status of RMS RAB fields when you call the OPEN procedure. If a field is not included in the following tables, it is initialized to zero.

Table 5–3: Setting of RMS Record Access Block Fields by a Call to OPEN

Field	Name	OPEN Parameters and Value
RAB$L_CTX	Context	Reserved to Digital.
RAB$L_FAB[1]	FAB address	Set to address of FAB (allocated by VAX Pascal RTL).
RAB$W_ISI	Internal stream identifier	Returned by RMS.
RAB$L_KBF	Key buffer address	May be modified for individual file operations after the file is opened.
RAB$B_KRF	Key of reference	May be modified for individual file operations after the file is opened.
RAB$B_KSZ	Key size	May be modified for individual file operations after the file is opened.
RAB$B_RAC	Record access mode	May be modified for individual file operations after the file is opened.
RAB$L_RBF	Record address	May be modified for individual file operations after the file is opened.
RAB$L_RHB	Record header buffer	Set to address of 2-byte carriage-control information for terminal files enabled for prompting.
RAB$L_ROP	Record options	
RAB$V_NLK	No lock	May be modified for individual file operations after the file is opened.
RAB$V_RAH	Read ahead	1.
RAB$V_TPT	Truncate file often PUT	May be modified for individual file operations after the file is opened.
RAB$V_UIF	Update if record exists	1, if ACCESS:=DIRECT.
RAB$V_WBH	Write behind	1.
RAB$W_RSZ	Record size	May be modified for individual file operations after the file is opened.
RAB$L_STS	Completion status code	Returned by RMS.
RAB$L_UBF[1]	User record area address	Set to buffer address after file is opened (VAX Pascal RTL allocates buffer).

[1]After the call to OPEN, this field must contain the same value it had before the call.

(continued on next page)

Table 5–3 (Cont.): Setting of RMS Record Access Block Fields by a Call to OPEN

Field	Name	OPEN Parameters and Value
RAB$W_USZ[1]	User record area size	Set to size of record area; for files other than TEXT, the size is equal to the size of the component type; for TEXT files, the size is equal to the value of RECORD_LENGTH; otherwise, 255.

[1]After the call to OPEN, this field must contain the same value it had before the call.

Table 5–4 presents the status of RMS XAB fields when you call the OPEN procedure. If a field is not included in the following tables, it is initialized to zero.

Table 5–4: Setting of Extended Attribute Block Fields by a Call to OPEN

Field	Name	PASCAL OPEN Keyword and Value
XAB$B_DTP	Data type of key	Set to data type of key
XAB$B_FLG	Key option flags	
XAB$V_CHG	Changes allowed	0 if key is 0, else 1
XAB$V_DUP	Duplicates allowed	0 if key is 0, else 1
XAB$W_POS0	Key position	Position of key in indexed file
XAB$B_REF	Key of reference	Primary key is 0, first alternate key is 1, second alternate key is 2, and so on
XAB$B_SIZ0	Key size	Size of key

Table 5–5 presents the status of RMS Name Block fields when you call the OPEN procedure. If a field is not included in the following tables, it is initialized to zero.

Table 5–5: Setting of Name Block Fields by a Call to OPEN

Field	Name	OPEN Keyword and Value
NAM$L_ESA[1]	Expanded string area	Address of RTL buffer

[1]These fields are only valid during execution of user-action functions; VAX Pascal places 0 in these fields after the call to OPEN.

(continued on next page)

Table 5–5 (Cont.): Setting of Name Block Fields by a Call to OPEN

Field	Name	OPEN Keyword and Value
NAM$B_ESS[1]	Expanded string area	NAM$C_MAXRSS
NAM$L_RSA	Expanded string area	Address of RTL buffer
NAM$B_RSS	Expanded string area	NAM$C_MAXRSS

[1]These fields are only valid during execution of user-action functions; VAX Pascal places 0 in these fields after the call to OPEN.

For More Information:

For information on opening indexed files, see Section 5.1.1.

5.1.7 Default Line Limits

VAX Pascal determines a default line limit for TEXT files by translating the logical name PAS$LINELIMIT as a string of decimal digits. If this logical name has not been defined, there is no default line limit. You can override the default by calling the LINELIMIT procedure.

For More Information:

For information on LINELIMIT, see the *VAX Pascal Reference Manual.*

5.2 User Action Functions

The user action parameter of the OPEN procedure allows you to access RMS facilities not explicitly available in the VAX Pascal language by writing a function that controls the opening of the file. Inclusion of the user action parameter causes the Run-Time Library to call your function to open the file instead of calling RMS to open it according to its normal defaults.

The user action parameter of the CLOSE procedure is similar to that of the OPEN procedure. It allows you to access RMS facilities not directly available in VAX Pascal by writing a function that controls the closing of the file. Including the user action parameter causes the Run-Time Library to call your function to close the file instead of calling RMS to close it according to its normal defaults.

When an OPEN or CLOSE procedure is executed, the Run-Time Library uses the procedure's parameters to establish the RMS file access block (FAB) and the record access block (RAB), as well as to establish its own internal data structures. These blocks are used to transmit requests for file and record operations to RMS; they are also used to return the data contents of files, information about file characteristics, and status codes.

In order, the three parameters passed to a user action function by the Run-Time Library are as follows:

- FAB address
- RAB address
- File variable

A **user action function** is usually written in VAX Pascal and includes the following:

- Modifications to the FAB or RAB, or both (optional)
- $OPEN and $CONNECT for existing files or $CREATE and $CONNECT for new files (required)
- Status check of the values returned by $OPEN or $CREATE and $CONNECT (required)
- Storage of FAB and RAB values in program variables (optional)
- Return of success or failure status value for the user action function (required)

NOTE

Modification of any of the RMS file access blocks provided by the Run-Time Library may interfere with the normal operation of the VAX Pascal Run-Time Library.

Example 5–1 shows a VAX Pascal program that copies one file into another. The program features two user action functions, which allow the output file to be created with the same size as the input file and to be given contiguous allocation on the storage media.

Example 5–1: User Action Function

```
[INHERIT( 'SYS$LIBRARY:STARLET' )]
PROGRAM Contiguous_Copy( F_In, F_Out );

{
The input file F_In is copied to the output file F_Out. F_Out has
the same size as F_In and has contiguous allocation.
}

TYPE
   FType = FILE OF VARYING[133] OF CHAR;

VAR
   F_In, F_Out    : FType;
   Alloc_Quantity : UNSIGNED;

FUNCTION User_Open( VAR FAB : FAB$TYPE;
                    VAR RAB : RAB$TYPE;
                    VAR F   : FType) : INTEGER;
   VAR
     Status : INTEGER;
   BEGIN          { Function User_Open }
   {Open file and remember allocation quantity }
   Status := $OPEN( FAB );
   IF ODD( Status ) THEN
      Status := $CONNECT( RAB );
   Alloc_Quantity := FAB.FAB$L_ALQ;
   User_Open := Status;
   END;           { Function User_Open }

FUNCTION User_Create( VAR FAB : FAB$TYPE;
                      VAR RAB : RAB$TYPE;
                      VAR F   : FType ) : INTEGER;
   VAR
     Status : INTEGER;
   BEGIN          { Function User_Create }
   { Set up contiguous allocation }
   FAB.FAB$L_ALQ := Alloc_Quantity;
   FAB.FAB$V_CBT := FALSE;
   FAB.FAB$V_CTG := TRUE;
   Status := $CREATE( FAB );
   IF ODD( Status ) THEN
      Status := $CONNECT( RAB );
   User_Create := Status;
   END;           { Function User_Create }

BEGIN             { main program }
{ Open files }
OPEN( F_In, HISTORY := READONLY, USER_ACTION := User_Open );
RESET(F_In );
OPEN( F_Out, HISTORY := NEW, USER_ACTION := User_Create );
REWRITE( F_Out );
```

(continued on next page)

Example 5–1 (Cont.): User Action Function

```
{ Copy F_In to F_Out }
WHILE NOT EOF( F_In ) DO
   BEGIN
   WRITE( F_Out, F_In_^ );
   GET( F_In );
   END;

{ Close files }
CLOSE( F_In );
CLOSE( F_Out );
END.                { main program }
```

In this example, the record types FAB$TYPE and RAB$TYPE are defined in SYS$LIBRARY:STARLET, which the program inherits. The function User_Open is called as a result of the OPEN procedure for the input file F_In. The function begins by opening the file with the RMS service $OPEN. If $OPEN succeeds, the value of Status is odd; in that case, $CONNECT is performed. The allocation quantity contained in the FAB.FAB$L_ALQ field of the FAB is assigned to a variable so that this value can be used in the second user action function. User_Open is then assigned the value of Status (in this case, TRUE), which is returned to the main program.

The function User_Create is called as a result of the OPEN procedure for the output file F_Out. The function assigns the allocation quantity of the input file to the FAB.FAB$L_ALQ field of the FAB, which contains the allocation size for the output file. The FAB field FAB.FAB$V_CBT is set to FALSE to disable the request that file storage be allocated contiguously on a best try basis. Then, the FAB field FAB.FAB$V_CTG is set to TRUE so that contiguous storage allocation is mandatory. Finally, the RMS service $CREATE is performed. If $CREATE is successful, $CONNECT will be done and the function return value will be that of $CREATE.

Once the OPEN procedures have been performed successfully, the program can then accomplish its main task, copying the input file F_In to the output file F_Out, which is the same size as F_In and has contiguous allocation. The last step in the program is to close the files.

For More Information:

- On the OPEN and CLOSE procedures (*VAX Pascal Reference Manual*)
- On RMS file access blocks (Section 5.1.6.2)

5.3 File Sharing

Through the RMS file sharing capability, a file can be accessed by more than one open program at a time or by the same program through more than one file variable. There are two kinds of file sharing: read sharing and write sharing. Read sharing occurs when several programs are reading a file at the same time. Write sharing takes place when at least one program is writing a file and at least one other program is either reading or writing the same file.

The extent to which file sharing can take place is determined by the following factors:

- Device type

  Sharing is possible only on disk files, since other files must be accessed sequentially.

- File organization

  All three file organizations permit read and write sharing on disk files.

- Explicit user-supplied information

  Whether or not file sharing actually takes place depends on two items of information that you provide for each program accessing the file. In VAX Pascal programs, this information is supplied by the values of the SHARING and HISTORY parameters in the OPEN procedure.

The HISTORY parameter determines how the program will access the file. HISTORY := NEW, HISTORY := OLD, and HISTORY := UNKNOWN determine that the program will read from and write to the file. HISTORY := READONLY determines that the program will only read from the file. If you try to open an existing file with HISTORY := OLD or HISTORY := UNKNOWN, the Run-time Library retries the OPEN procedure with HISTORY := READONLY if the initial OPEN fails with a privilege violation.

The SHARING parameter determines what other programs are allowed to do with the file. Read sharing can occur when SHARING := READONLY is specified by all programs that access the file. Write sharing is accomplished when all programs specify SHARING := READWRITE. To prevent sharing, specify SHARING:= NONE with the first program to access the file.

Programs that specify SHARING := READONLY or SHARING := READWRITE can access a file simultaneously; however, file sharing can fail under certain circumstances. For example, a program without either of these parameters will fail when it attempts to open a file currently being accessed by some other program. Or, a program that specifies SHARING :=

READONLY or SHARING := READWRITE can fail to open a file because a second program with a different specification is currently accessing that file.

When two or more programs are write sharing a file, each program should include a condition handler. This error-processing mechanism prevents program failure due to a record-locking error.

For More Information:

- On record-locking errors (Section 5.4)
- On condition handling (Chapter 7)
- On the OPEN procedure (*VAX Pascal Reference Manual*)

5.4 Record Locking

The RMS record locking facility, along with the logic of the program, prevents two processes from accessing the same component simultaneously. It ensures that a program can add, delete, or update a component without having to do a synchronization check to determine whether that component is currently being accessed by another process.

When a program opens a relative or indexed file and specifies SHARING := READWRITE, RMS locks each component as it is accessed. When a component is locked, any program that attempts to access it fails and a record-locked error results. A subsequent I/O operation on the file variable unlocks the previously accessed component. Thus, at most one component is locked for each file variable.

If you use the READ procedure, VAX Pascal will implicitly unlock the component by executing the UNLOCK procedure during the execution of the READ procedure.

A VAX Pascal program can explicitly unlock a component by executing the UNLOCK procedure. To minimize the time during which a component is locked against access by other programs, the UNLOCK procedure should be used in programs that retrieve components from a shared file but that do not attempt to update them. VAX Pascal requires that a component be locked before a DELETE or an UPDATE procedure can be executed.

For More Information:

For information on the OPEN, UNLOCK, DELETE, and UPDATE procedures, see *VAX Pascal Reference Manual*.

Chapter 6

Implementation Notes: Predeclared Routines and Attributes

This chapter provides notes on environment-specific dependencies or information about the VAX Pascal predeclared routines and attributes, and contains the following sections:

- Predeclared routines (Section 6.1)
- Attributes (Section 6.2)

If this chapter contains no information on a predeclared routine or an attribute, then this environment supports the element exactly as it is described in the *VAX Pascal Reference Manual*.

6.1 Environment-Specific Information on Predeclared Routines

The following sections describe environment-specific dependencies or information about the VAX Pascal predeclared routines. All of the attributes described in *VAX Pascal Reference Manual* are supported.

6.1.1 ADD_INTERLOCKED Function

This function is implemented using the VAX Add Aligned Word Interlocked (ADAWI) instruction.

6.1.2 CLEAR_INTERLOCKED Function

This routine is implemented using the VAX Branch on Bit Clear and Clear Interlocked (BBCCI) instruction.

6.1.3 DATE Function

This function is implemented using the LIB$FORMAT_DATE_TIME routine.

6.1.4 GETTIMESTAMP Procedure

The last two components of an object of type TIMESTAMP are BINARY_TIME and DAY_OF_WEEK. The BINARY_TIME field is the 64-bit VMS binary time. The DAY_OF_WEEK field is the value from the LIB$DAY_OF_WEEK routine. GETTIMESTAMP is implemented using the LIB$CONVERT_DATE_STRING routine.

6.1.5 HALT Procedure

This routine calls the VMS Run-Time Library procedure LIB$STOP with the condition value PAS$_HALT.

6.1.6 MFPR Function

When you call this function, the value of the internal processor register is retrieved with the MFPR privileged VAX instruction.

NOTE

The VAX Pascal compiler generates user-mode code. VAX Pascal does not explicitly support the running of VAX Pascal generated code in kernel mode. However, if the following rules are observed, then the generated code has a good chance of working as expected in elevated access modes:

- All code must be compiled with the /NOCHECK qualifier or [CHECK(NONE)] attribute. The VAX Pascal run-time signaling method relies on trying to execute the HALT instruction. In user-mode, this causes an exception which is a signal to the VAX Pascal Run-Time Library. In kernel-mode, this simply HALTs the machine.

- Avoid all routine calls which translate into Run-Time Library calls. These include all I/O routines, several arithmetic routines, several string routines, and so forth.

6.1.7 MTPR Procedure

When you call this procedure, the source expression is moved into the internal processor register with the MTPR privileged VAX instruction.

For More Information:

For information on running VAX Pascal code in kernel mode, see Section 6.1.6.

6.1.8 SET_INTERLOCKED Function

This routine is implemented using the VAX Branch on Bit Set and Set Interlocked (BBSSI) instruction.

6.1.9 TIME Function

This function is implemented using the LIB$FORMAT_DATE_TIME routine.

6.2 Environment-Specific Information on Attributes

The following sections describe environment-specific dependencies or information about the VAX Pascal attributes. All of the attributes described in *VAX Pascal Reference Manual* are supported.

6.2.1 ALIGNED Attribute

The value of ALIGNED(9) specifies page alignment on VAX machines. The value 9 is the largest number allowed.

6.2.2 INITIALIZE Attribute

VAX Pascal implements the INITIALIZE attribute using the LIB$INITIALIZE routine.

For More Information:

For information on using the LIB$INITIALIZE routine, see the *VMS Run-Time Library Routines Volume*.

6.2.3 INHERIT Attribute

VAX Pascal allows 512 environment files to be used in a compilation unit.

Chapter 7

Error Processing and Condition Handling

An exception condition is an event, usually an error, that occurs during program execution and is detected by system hardware or software or the logic in a user application program. A **condition handler** is a routine that is used to resolve exception conditions.

By default, the VAX Condition Handling Facility (CHF) provides condition handling sufficient for most VAX Pascal programs. The CHF also processes user-written condition handlers.

This chapter discusses the following topics:

- Condition handling terms (Section 7.1)
- Overview of condition handling (Section 7.2)
- Writing condition handlers (Section 7.3)
- Fault and trap handling (Section 7.4)

The use of condition handlers requires considerable programming experience and should not be undertaken by novice users. You should understand the discussions of condition handling in the following volumes before attempting to write your own condition handler:

- *VMS Run-Time Library Routines Volume*
- *VMS System Services Volume*
- *Introduction to VMS System Routines*

7.1 Condition Handling Terms

The following terms are used in the discussion of condition handling:

- Condition value—An integer value that identifies a specific condition.
- Stack frame—A standard data structure built on the stack during a routine call, starting from the location addressed by the frame pointer (FP) and proceeding to both higher and lower addresses; it is popped off the stack during the return from a routine.
- Routine activation—The environment in which a routine executes. This environment includes a unique stack frame on the run-time stack; the stack frame contains the address of a condition handler for the routine activation. A new routine activation is created every time a routine is called and is deleted when control passes from the routine.
- Establish—The process of placing the address of a condition handler in the stack frame of the current routine activation. A condition handler established for a routine activation is automatically called when a condition occurs. In VAX Pascal, condition handlers are established by means of the predeclared procedure ESTABLISH. A routine that establishes a condition handler is known as an *establisher*.
- Program exit status—The status of the program at its completion.
- Signal—The means by which the occurrence of an exception condition is made known. Signals are generated by the operating system in response to I/O events and hardware errors, by the system-supplied library routines, and by user routines. All signals are initiated by a call to the signaling facility, for which there are two entry points:
  - LIB$SIGNAL—Used to signal a condition and, possibly, to continue program execution
  - LIB$STOP—Used to signal a severe error and discontinue program execution, unless a condition handler performs an unwind operation
- Resignal—The means by which a condition handler indicates that the signaling facility is to continue searching for a condition handler to process a previously signaled error. To resignal, a condition handler returns the value SS$_RESIGNAL.
- Unwind—The return of control to a particular routine activation, bypassing any intermediate routine activations. For example, if X calls Y, and Y calls Z, and Z detects an error, then a condition handler associated with X or Y can unwind to X, bypassing Y. Control returns to X immediately following the point at which X called Y.

7.2 Overview of Condition Handling

When the VMS system creates a user process, a system-defined condition handler is established in the absence of any user-written condition handler. The system-defined handler processes errors that occur during execution of the user image. Thus, by default, a run-time error causes the system-defined condition handler to print error messages and to terminate or continue execution of the image, depending on the severity of the error.

When a condition is signaled, the system searches for condition handlers to process the condition. The system conducts the search for condition handlers by proceeding down the stack, frame by frame, until a condition handler is found that does not resignal. The default handler calls the system's message output routine to send the appropriate message to the user. Messages are sent to the SYS$OUTPUT and SYS$ERROR files. If the condition is not a severe error, program execution continues. If the condition is a severe error, the default handler forces program termination, and the condition value becomes the program exit status.

You can create and establish your own condition handlers according to the needs of your application. For example, a condition handler could create and display messages that describe specific conditions encountered during the execution of your program, instead of relying on system error messages.

7.2.1 Condition Signals

A condition signal consists of a call to either LIB$SIGNAL or LIB$STOP, the two entry points to the signaling facility. These entry points can be inherited from SYS$LIBRARY:PASCAL$LIB_ROUTINES.PEN.

If a condition occurs in a routine that is not prepared to handle it, a signal is issued to notify other active routines. If the current routine can continue after the signal is propagated, you can call LIB$SIGNAL. A higher-level routine can then determine whether program execution should continue. If the nature of the condition does not allow the current routine to continue, you can call LIB$STOP.

7.2.2 Handler Responses

A condition handler responds to an exception condition by taking action in three major areas:

- Condition correction
- Condition reporting
- Execution control

The handler first determines whether the condition can be corrected. If so, it takes the appropriate action, and execution continues. If the handler cannot correct the condition, the condition may be resignaled; that is, the handler requests that another condition handler be sought to process the condition.

A handler's condition reporting can involve one or more of the following actions:

- Maintaining a count of exceptions encountered during program execution
- Resignaling the same condition to send the appropriate message to the output file
- Changing the severity field of the condition value and resignaling the condition
- Signaling a different condition, for example, the production of a message designed for a specific application

A handler can control execution in several ways:

- By continuing from the signal. If the signal was issued through a call to LIB$STOP, the program exits.
- By doing a nonlocal GOTO operation (see Section 7.5, Example 5).
- By unwinding to the establisher at the point of the call that resulted in the exception. The handler can then determine the function value returned by the called routine.
- By unwinding to the establisher's caller (the routine that called the routine which established the handler). The handler can then determine the function value returned by the called routine.

7.3 Writing Condition Handlers

The following sections describe how to write and establish condition handlers and provide some simple examples.

7.3.1 Establishing and Removing Handlers

To use a condition handler, you must first declare the handler as a routine in the declaration section of your program; then, within the executable section, you must call the predeclared procedure ESTABLISH. The ESTABLISH procedure sets up a VAX Pascal language-specific condition handler that in turn allows your handler to be called. User-written condition handlers set up by ESTABLISH must have the ASYNCHRONOUS attribute and two integer array formal parameters. Such routines can access only local, read-only, and volatile variables, and local, predeclared, and asynchronous routines.

Because condition handlers are asynchronous, any attempt to access a non-read-only or nonvolatile variable declared in an enclosing block will result in a warning message. The predeclared file variables INPUT and OUTPUT are such nonvolatile variables; therefore, simultaneous access to these files from both an ordinary program and from an asynchronous condition handler's activation may have undefined results. The following steps outline the recommended method for performing I/O operations from a condition handler:

1. Declare a file with the VOLATILE attribute at program level.
2. Open this file to refer to SYS$INPUT, SYS$OUTPUT, or another appropriate file.
3. Use this file in the condition handler.

External routines (including system services) that are called by a condition handler require the ASYNCHRONOUS attribute in their declaration.

You should set up a user-written condition handler with the predeclared procedure ESTABLISH rather than with the Run-Time Library routine LIB$ESTABLISH. ESTABLISH follows the VAX Pascal procedure-calling rules and is able to handle VAX Pascal condition handlers more efficiently than LIB$ESTABLISH. A condition handler set up by LIB$ESTABLISH might interfere with the default error handling of the VAX Pascal run-time system, and cause unpredictable results.

The following example shows how to establish a condition handler using the VAX Pascal procedure ESTABLISH:

```
[EXTERNAL,ASYNCHRONOUS] FUNCTION Handler
   (VAR Sigargs  : Sigarr;
    VAR Mechargs : Mecharr) : INTEGER;
   EXTERN;
   .
   .
   .
ESTABLISH (Handler);
```

To establish the handler, call the ESTABLISH procedure. To remove an established handler, call the predeclared procedure REVERT, as follows:

```
REVERT;
```

As a result of this call, the condition handler established in the current stack frame is removed. When control passes from a routine, any condition handler established during the routine's activation is automatically removed.

7.3.2 Declaring Parameters for Condition Handlers

A VAX Pascal condition handler is an integer-valued function that is called when a condition is signaled. Two formal VAR parameters must be declared for a condition handler:

- An integer array to refer to the parameter list from the call to the signal routine (the signal array); that is, the list of parameters included in calls to LIB$SIGNAL or LIB$STOP (see Section 7.2.1)
- An integer array to refer to information concerning the routine activation that established the condition handler (the mechanism array)

For example, a condition handler can be defined as follows:

```
TYPE
   Sigarr  = ARRAY[0..9] OF INTEGER;
   Mecharr = ARRAY[0..4] OF INTEGER;

[EXTERNAL,ASYNCHRONOUS] FUNCTION Handler
   (VAR Sigargs : Sigarr;
    VAR Mechargs : Mecharr)    : INTEGER;
   EXTERN;
      .
      .
      .
ESTABLISH (Handler);
      .
      .
      .
```

The signal procedure passes the following values to the array Sigargs:

Value	Description
Sigargs[0]	The number of parameters being passed in this array (parameter count).
Sigargs[1]	The primary condition being signaled (condition value). See Section 7.3.4 for a discussion of condition values.
Sigargs[2 to n]	The optional parameters supplied in the call to LIB$SIGNAL or LIB$STOP; note that the index range of Sigargs should include as many entries as are needed to refer to the optional parameters.

The routine that established the condition handler passes the following values, which contain information about the establisher's routine activation, to the array Mechargs:

Value	Description
Mechargs[0]	The number of parameters being passed in this array.
Mechargs[1]	The address of the stack frame that established the handler.
Mechargs[2]	The number of calls that have been made (that is, the stack frame depth) from the routine activation up to the point at which the condition was signaled.
Mechargs[3]	The value of register R0 at the time of the signal.
Mechargs[4]	The value of register R1 at the time of the signal.

7.3.3 Handler Function Return Values

Condition handlers are functions that return values to control subsequent execution. These values and their effects are listed as follows:

Value	Effect
SS$_CONTINUE	Continues execution from the signal. If the signal was issued by a call to LIB$STOP, the program does not continue, but exits.
SS$_RESIGNAL	Resignals to continue the search for a condition handler to process the condition.

In addition, a condition handler can request a stack unwind by calling the $UNWIND system service routine. You can inherit $UNWIND from SYS$LIBRARY:STARLET.PEN.

When $UNWIND is called, the function return value of the condition handler is ignored. The handler modifies the saved registers R0 and R1 in the mechanism parameters to specify the called function's return value.

A stack unwind is usually made to one of two places:

- The point in the establisher at which the call was made that resulted in the exception. Specify the following:

```
Status := $UNWIND (Mechargs[2],0);
```

- The routine that called the establisher. Specify the following:

```
Status := $UNWIND (Mechargs[2]+1,0);
```

7.3.4 Condition Values and Symbols

The VMS system uses condition values to indicate that a called routine has either executed successfully or failed, and to report exception conditions. Condition values are usually symbolic names that represent 32-bit packed records, consisting of fields (usually interpreted as integers) that indicate which system component generated the value, the reason the value was generated, and the severity of the condition.

A warning severity code (0) indicates that although output was produced, the results may be unpredictable. An error severity code (2) indicates that output was produced even though an error was detected. Execution can continue, but the results may not be correct. A severe error code (4) indicates that the error was of such severity that no output was produced.

A condition handler can alter the severity code of a condition value to allow execution to continue or to force an exit, depending on the circumstances.

Occasionally a condition handler may require a particular condition to be identified by an exact match; that is, each bit of the condition value bits (0..31) must match the specified condition. For example, you may want to process a floating overflow condition only if the severity code is still 4 (that is, if no previous condition handler has changed the severity code) and the control bits have not been modified. A typical condition handler response is to change the severity code and resignal.

In most cases, however, you want some response to a condition, regardless of the value of the severity code or control bits. To ignore the severity and control fields of a condition value, declare and call the LIB$MATCH_COND function.

For More Information:

- On the format of a condition value (*Introduction to VMS System Routines*)
- On calling the LIB$MATCH_COND function (Section 7.5)

7.4 Fault and Trap Handling

If a VAX processor detects an error while executing a machine instruction, it can take one of two actions. The first action, called a fault, preserves the contents of registers and memory in a consistent state so that the instruction can be restarted. The second action, called a trap, completes the instruction, but with a predefined result. For example, if an integer overflow trap occurs, the result is the correct low-order part of the true value.

The action taken when an exception occurs depends on the type of exception. For example, faults occur for access violations and for detection of a floating reserved operand. Traps occur for integer overflow and for integer divide-by-zero exceptions. However, when a floating overflow, floating underflow, or floating divide-by-zero exception occurs, the action taken depends on the type of VAX processor executing the instruction. The original VAX–11/780 processor traps when these errors occur and stores a floating reserved operand in the destination. All other VAX processors fault on these exceptions, allowing the error to be corrected and the instruction restarted.

If your program is written to handle floating traps, but runs on a VAX processor that generates faults, execution may continue incorrectly. For example, if a condition handler merely causes execution to continue after a floating trap, a reserved operand is stored and the next instruction is executed. However, the same handler used on a processor that generates faults causes an infinite loop of faults because it restarts the erroneous instruction. Therefore, you should write floating-point exception handlers that take the appropriate actions for both faults and traps.

Separate sets of condition values are signaled by the processor for faults and traps. Exceptions and their condition code names are as follows:

Exception	Fault	Trap
Floating overflow	SS$_FLTOVF_F	SS$_FLTOVF
Floating underflow	SS$_FLTUND_F	SS$_FLTUND
Floating divide-by-zero	SS$_FLTDIV_F	SS$_FLTDIV

To convert faults to traps, you can use the Run-Time Library LIB$SIM_TRAP procedure either as a condition handler or as a called routine from a user-written handler. When LIB$SIM_TRAP recognizes a floating fault, it simulates the instruction completion as if a floating trap had occurred.

7.5 Examples of Condition Handlers

The examples in this section inherit the $UNWIND system service routine from SYS$LIBRARY:STARLET.PEN. They also assume the following declaration has been made:

```
[INHERIT( 'SYS$LIBRARY:STARLET', 'SYS$LIBRARY:PASCAL$LIB_ROUTINES' )]
PROGRAM Error_Handling( INPUT, OUTPUT);

TYPE
   Sig_Args  = ARRAY[0..100] OF INTEGER;        { Signal parameters }
   Mech_Args = ARRAY[0..4] OF [UNSAFE] INTEGER; { Mechanism parameters }
```

Example 1

```
[ASYNCHRONOUS] FUNCTION Handler_0
   (VAR SA : Sig_Args;
    VAR MA : Mech_Args)    : [UNSAFE] INTEGER;

   BEGIN
   IF LIB$MATCH_COND (SA[1],   condition-name   ,...) <> 0
   THEN
      BEGIN
          .
          .                          { do something appropriate }
          .
      Handler_0 := SS$_CONTINUE;   { condition handled,
                                     propagate no further }
      END
   ELSE
      Handler_0 := SS$_RESIGNAL;   { propagate condition
                                     status to other handlers }
   END;
```

This example shows a simple condition handler. The handler identifies the condition being signaled as one that it is prepared to handle and then takes appropriate action. Note that for all unidentified condition statuses, the handler resignals. A handler must always follow this behavior.

Example 2

```
[ASYNCHRONOUS] FUNCTION Handler_1
   (VAR SA : Sig_Args;
    VAR MA : Mech_Args)    : [UNSAFE] INTEGER;

   BEGIN
   IF SA[1] = SS$_UNWIND
   THEN
      BEGIN
          .
          .                              { cleanup }
          .
      END;
   Handler_1 := SS$_RESIGNAL;
   END;
```

When writing a handler, remember that it can be activated with a condition of SS$_UNWIND, signifying that the establisher's stack frame is about to be unwound. If the establisher has special cleanup to perform, such as freeing dynamic memory, closing files, or releasing locks, the handler should check for the SS$_UNWIND condition status. If there is no cleanup, the required action of resignaling all unidentified conditions results in the correct behavior. On return from a handler activated with SS$_UNWIND, the stack frame of the routine that established the handler is deleted (unwound).

Example 3

```
[ASYNCHRONOUS] FUNCTION Handler_2
   (VAR SA : Sig_Args;
    VAR MA : Mech_Args)    : [UNSAFE] INTEGER;

   BEGIN
   IF LIB$MATCH_COND (SA[1],   condition-name   ,...) <> 0
   THEN
      BEGIN
          .
          .                          { cleanup }
          .
      MA[3] := expression;           { establish function result
                                       seen by caller }
     $UNWIND;                        { unwind to caller
                                       of establisher }
     END;
   Handler_2 := SS$_RESIGNAL;
   END;
```

A handler can perform a default unwind to force return to the caller of its establisher. If the establisher is a function whose result is expected in R0 or R0 and R1, the handler must establish the return value by modifying the third or third and fourth positions of the mechanism array (the locations of the return R0 and R1 values). If the establisher is a function whose result is returned by the extra-parameter method, the handler must establish the condition value by assignment to the function identifier. In this case, you must observe two additional restrictions:

- The handler must be nested within the function
- The function result must be declared VOLATILE

Example 4

```
[ASYNCHRONOUS] FUNCTION Handler_3
   (VAR SA : Sig_Args;
    VAR MA : Mech_Args)     : [UNSAFE] INTEGER;

   BEGIN
   IF LIB$MATCH_COND (SA[1],   condition-name   ,...) <> 0
   THEN
      BEGIN
          .
          .                          { cleanup }
          .
      MA[3] := expression;           { establish function result
                                       seen by caller }
      $UNWIND (MA[2]);               { unwind to establisher }
      END;
   Handler_3 := SS$_RESIGNAL;
   END;
```

A handler can also force return to its establisher immediately following the point of call. In this case, you should make sure that the handler understands whether the currently uncompleted call was a function call (in which case a returned value is expected) or a procedure call. If the uncompleted call is a function call that will return a value in R0 or R0 and R1, then the handler can modify the mechanism array to supply a value. If, however, the uncompleted call is a function call that will return a value using the extra-parameter mechanism, then there is no way for the handler to supply a value.

Example 5

```
[ASYNCHRONOUS] FUNCTION Handler_4
   (VAR SA : Sig_Args;
    VAR MA : Mech_Args)     : [UNSAFE] INTEGER;

   BEGIN
   IF LIB$MATCH_COND (SA[1],   condition-name   ,...) <> 0
   THEN
      GOTO 99;
   Handler_4 := SS$_RESIGNAL;
   END;
```

A handler can force control to resume at an arbitrary label in its scope. Note that this reference is to a label in an enclosing block, because a GOTO to a local label would simply remain within the handler. In accordance with the VAX Procedure Calling Standard, VAX Pascal implements references to labels in enclosing blocks by signaling SS$_UNWIND in all stack frames that must be deleted.

Example 6

```
FUNCTION EXP_With_Status
   (X : REAL;
    VAR Status : INTEGER )     : REAL;

   FUNCTION MTH$EXP
      (A : REAL) : REAL;
      EXTERNAL;

   [ASYNCHRONOUS] FUNCTION Math_Error
      (VAR SA : Sig_Args;
       VAR MA : Mech_Args)        : [UNSAFE] INTEGER;

      BEGIN    { Math_Error }
      IF LIB$MATCH_COND (SA[1], MTH$_FLOOVEMAT, MTH$_FLOUNDMAT) <> 0
      THEN
         BEGIN
            IF ODD( Status )                  { record condition status
            THEN                                if no previous error }

            Status := SA[1]::Cond_Status;     { condition handled,
            Math_Error := SS$_CONTINUE;         propagate no further }
         END
       ELSE
          Math_Error := SS$_RESIGNAL;         { propagate condition status
                                                to other handlers }
       END;

   BEGIN    { EXP_With_Status }
   STATUS := SS$_SUCCESS;
   ESTABLISH (Math_Error);
   EXP_With_Status := MTH$EXP (X);
   END;
```

This example shows a handler that records the condition status if a floating overflow or underflow error is detected during the execution of the mathematical function MTH$EXP.

Example 7

```
[INHERIT('SYS$LIBRARY:STARLET')]
PROGRAM Use_A_Handler(INPUT,OUTPUT);

TYPE
   Sigarr  = ARRAY [0..9] OF INTEGER;
   Mecharr = ARRAY [0..4] OF INTEGER;

VAR
   F1,F2 : REAL;
[ASYNCHRONOUS] FUNCTION My_Handler
   (VAR Sigargs  : Sigarr;
    VAR Mechargs : Mecharr) : INTEGER;

   VAR
      Outfile : TEXT;

   [ASYNCHRONOUS] FUNCTION LIB$FIXUP_FLT
      (VAR Sigargs  : Sigarr;
       VAR Mechargs : Mecharr;
           New_Opnd : REAL := %IMMED 0) : INTEGER;
      EXTERNAL;

   [ASYNCHRONOUS] FUNCTION LIB$SIM_TRAP
      (VAR Sigargs  : Sigarr;
       VAR Mechargs : Mecharr) : INTEGER;
      EXTERNAL;
      BEGIN
      OPEN(Outfile,'TT:');
      REWRITE(Outfile);

      { Handle various conditions }
      CASE Sigargs[1] OF

      { Convert floating faults to traps }
      SS$_FLTDIV_F, SS$_FLTOVF_F :
         LIB$SIM_TRAP(Sigargs,Mechargs);

      { Handle the floating divide by zero trap }
      SS$_FLTDIV :
         BEGIN
            WRITELN(Outfile,'Floating divide by zero');
            My_Handler := SS$_CONTINUE;
         END;

      { Handle the floating overflow trap }
      SS$_FLTOVF :
         BEGIN
            WRITELN(Outfile,'Floating overflow');
            My_Handler := SS$_CONTINUE;
         END;

      { Handle taking the square root }
      MTH$_SQUROONEG :
          BEGIN
             WRITELN(Outfile,'Square root of a negative number');
             My_Handler := SS$_CONTINUE;
          END;
```

```
        { Handle the reserved operand left by SQRT }
        SS$_ROPRAND :
            BEGIN
               WRITELN(Outfile,'Reserved floating operand');
               LIB$FIXUP_FLT(Sigargs,Mechargs);
               My_Handler := SS$_CONTINUE;
            END;

        OTHERWISE
           BEGIN
              WRITELN(Outfile,'Condition occurred, ',HEX(Sigargs[1]));
              My_Handler := SS$_RESIGNAL;
           END;

        END;

        CLOSE(Outfile);

        END;
BEGIN
ESTABLISH(My_Handler);
F1 := 0.0;
F2 := 1E38;

{ Generate exception conditions }
F1 := F2 / 0.0;
F1 := F2 * f2;
F1 := SQRT(-1.0);
END.
```

This example demonstrates the use of VMS condition handlers with VAX Pascal.

Chapter 8

Programming Tools

This chapter provides information on tools that you can use to develop and to refine your VAX Pascal programs. Some of the products described ship with the VMS operating system, and other products must be purchased separately. The following products are described in this chapter:

- VMS Debugger (Section 8.1)
- VAX Text Processing Utility (Section 8.2)
- VAX Language-Sensitive Editor and VAX Source Code Analyzer (Section 8.3)
- VAX Common Data Dictionary (Section 8.4)

8.1 VMS Debugger

A debugger is a tool to help you locate run-time errors quickly. It enables you to observe and manipulate the program's execution interactively, step by step, until you locate the point at which the program stopped working correctly.

The VMS Debugger (provided with the VMS operating system) is a symbolic debugger. This means that you can refer to program locations by the symbols (names) you used for those locations in your program (the names of variables, routines, labels, and so on). You do not have to use virtual addresses to refer to memory locations.

If your program is written in more than one language, you can change from one language to another in the course of a debugging session. The current source language determines the format used for entering and displaying data, as well as other features that have language-specific settings (for example, comment characters, operators and operator precedence, and case sensitivity or insensitivity).

The following sections provide language-specific information on the VMS Debugger.

8.1.1 Compiling and Linking to Prepare for Debugging

The following example shows how to compile and link a VAX Pascal program (consisting of a single compilation unit named INVENTORY) so that subsequently you will be able to use the debugger:

```
$ PASCAL/DEBUG/NOOPTIMIZE INVENTORY
$ LINK/DEBUG INVENTORY
```

The /DEBUG qualifier on the Pascal command causes the compiler to write the debug symbol records associated with INVENTORY into the object module, INVENTORY.OBJ. These records allow you to use the names of variables and other symbols declared in INVENTORY with debugger commands. (If your program has several compilation units, you must compile each unit that you want to debug with the /DEBUG qualifier).

You should use the /NOOPTIMIZE qualifier when you compile in preparation for debugging. Without this qualifier, the resulting object code is optimized, possibly causing the contents of some program locations to be inconsistent with what you might expect from the source code. (After the program has been debugged, you will probably want to recompile it without the /NOOPTIMIZE qualifier, because optimization may reduce a program's size and increase the execution speed.)

The /DEBUG qualifier on the LINK command causes the linker to include all symbol information that is contained in INVENTORY.OBJ in the executable image. The qualifier also causes the VMS image activator to start the debugger at run time. (If your program has several object modules, you may need to specify other modules in the LINK command.)

8.1.2 Starting and Terminating a Debugging Session

Before you invoke the debugger, check the current debugger configuration, as follows:

```
$ SHOW LOGICAL DBG$PROCESS
%SHOW-S-NOTRAN, no translation for logical name DBG$PROCESS
```

If the DBG$PROCESS logical has a value other than undefined (as in the previous example) or DEFAULT, enter the following command to change this value:

```
$ DEFINE DBG$PROCESS DEFAULT
```

You invoke the debugger by issuing the DCL command RUN. The following message will appear on your screen:

```
$ RUN INVENTORY
                    VAX DEBUG Version 5.n

%DEBUG-I-INITIAL, language is PASCAL, module set to 'INVENTORY'
DBG>
```

The DBG> prompt indicates that you can now type debugger commands. At this point, if you type the GO command, program execution begins and continues until it is forced to pause or stop (for example, if the program prompts you for input, or an error occurs).

To interrupt a debugging session and return to the debugger prompt, press CTRL/C. This is useful if, for example, your program loops or you want to interrupt a debugger command that is still in progress. For example:

```
DBG> GO
   .
   .
   .
(infinite loop)
CTRL/C
Interrupt
%DEBUG-W-ABORTED, command aborted by user request
DBG>
```

The following message indicates that your program has completed successfully:

```
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG>
```

To end a debugging session, type the EXIT command at the DBG> prompt or press CTRL/Z, as follows:

```
DBG> EXIT
$
```

8.1.3 Notes on VAX Pascal Support

In general, the VMS Debugger supports the data types and operators of VAX Pascal and of the other debugger-supported languages. However, there are important language-specific limitations. (To get information on the supported data types and operators of any of the languages, type the HELP LANGUAGE command at the DBG> prompt.)

In general, you can examine, evaluate, and deposit into variables, record fields, and array components. An exception to this occurs under the following circumstances: if a variable is not referenced in a program, the VAX Pascal compiler may not allocate the variable. If the variable is not allocated and you try to examine it or deposit into it, you will receive an error message.

When depositing data into variables, the debugger truncates the high-order bits if the value being deposited is larger than the variable; it fills the high-order bits with zeros if the value being deposited is smaller than the variable. If the deposit violates the rules of assignment compatibility, the debugger displays an informational message.

Automatic variables (within any active block) can be examined and can have values deposited into them; however, since automatic variables are allocated in stack storage and are contained in registers, their values are considered undefined until the variables are initialized or assigned a value. For example:

```
DBG> EXAMINE X
MAINP\X: 2147287308
```

In this example, the value of variable X should be considered undefined until after a value has been assigned to X.

In addition, although you may examine a VARYING OF CHAR string, it is not possible to examine the LENGTH field. For example, the following is not supported:

```
DBG> EXAMINE VARY_STRING.LENGTH
```

Because the current LENGTH of a VARYING string is the first word, you should do the following to examine the LENGTH:

```
DBG> EXAMINE/WORD VARY_STRING
```

It should also be noted that the typecast operator (::) is not permitted when evaluating VAX Pascal expressions.

8.1.4 Sample Debugging Session

The following example shows a program Sum, with the line numbers assigned by the compiler. This program prompts for a number and prints the sum of integers from 1 through the number entered. The error in program Sum is obvious, the value of the variable Total is not reset to 0 when a new number is entered, but the example illustrates some simple debugging commands.

```
1       PROGRAM Sum (INPUT, OUTPUT);
2
3       VAR
4          I, Highest, Total : INTEGER;
5
6       BEGIN
7       Total := 0;
8       WRITE ('Enter an integer; type 0 to quit : ');
9       READ (Highest);
10      WHILE Highest <> 0 DO
11         BEGIN
12         FOR I := 1 TO Highest DO
13            BEGIN
14            Total := Total + I;
15            END;
16         WRITELN ('The sum of integers from 1 to ', Highest:3,
17            ' is ', Total:4);
18         WRITE ('Enter an integer; type 0 to quit : ');
19         READ (Highest);
20         END;
21      END.
```

Initially, you would compile, link, and run the program as follows:

```
$ PASCAL SUM
$ LINK SUM
$ RUN SUM
Enter an integer; type 0 to quit :  5
The sum of integers from 1 to   5 is   15
Enter an integer; type 0 to quit :  4
The sum of integers from 1 to   4 is   25
Enter an integer; type 0 to quit :  0
$
```

The program returns a correct sum for the first number you enter, but the sum for the second number is too high. You compile and link to prepare for debugging as follows:

```
$ PASCAL/LIST/DEBUG/NOOPTIMIZE SUM
$ LINK/DEBUG SUM
$ PRINT SUM
```

The PRINT command prints the listing, which shows the compiler-generated line numbers. You are now ready to begin a debugging session. The callout numbers in the following terminal session are keyed to the notes that follow.

```
❶ $ RUN SUM

                    VAX DEBUG Version 5.x

  %DEBUG-I-INITIAL, language is PASCAL, module set to 'SUM'
❷ DBG> SET BREAK %LINE 9
  DBG> SET BREAK %LINE 19
```

```
❸ DBG>  GO
❹ break at SUM\%LINE 9
          9:         READ (Highest);
❺ DBG>  EXAMINE TOTAL
  SUM\TOTAL: 0
❻ DBG>  GO
  Enter an integer; type 0 to quit :  5
  The sum of integers from 1 to   5 is   15
  break at SUM\%LINE 19
❼         19:         READ (Highest);
❽ DBG>  EXAMINE TOTAL
  SUM\TOTAL: 15
❾ DBG>  DEPOSIT TOTAL=0
❿ DBG>  GO
  Enter an integer; type 0 to quit :  4
  The sum of integers from 1 to   4 is   10
  break at SUM\%LINE 19
          19:         READ (Highest);
  DBG>  GO
⓫ Enter an integer; type 0 to quit :  0
  %DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
⓬ DBG>  EXIT
  $
```

❶ When you enter the RUN command, the debugger displays an informational message and the DBG> prompt.

❷ You decide that the problem may lie with the initialization of Total. You can test this hypothesis by examining the value of Total each time you enter a new number. To do this, set two breakpoints, one at each READ procedure call that reads a number from the terminal.

❸ The GO command starts the execution of the program. The debugger tells you where in the program execution begins.

❹ When the first READ procedure call is reached, the debugger interrupts the program's execution and prompts you to enter a command.

❺ You examine the variable Total. Its value is 0, as expected at this point.

❻ The GO command continues the execution of the program, which prompts for a number. The program's response to the number you enter is correct.

❼ The debugger reaches the breakpoint at the second READ procedure call.

❽ You examine the variable Total. Its value is 15, not 0 as it should be. The value of Total needs to be reset to 0 before a new number is read.

❾ The DEPOSIT command replaces the contents of Total with 0. This deposit allows the program to return a correct result the next time the loop is executed.

❿ The GO command continues program execution. The result is correct.

⓫ When you enter a 0 to the prompt, the program exits. The debugger displays a message indicating the termination status.

⓬ The EXIT command terminates the debugging session.

You can now correct the program so that it initializes the variable Total correctly.

For More Information:

For information on the debugger, see the *VMS Debugger Manual*.

8.2 VAX Text Processing Utility (TPU)

The VAX Text Processing Utility (TPU) (provided with the VMS operating system) is a high-performance, programmable utility. TPU provides a number of special features, such as multiple buffers and windows, definable keys and key sequences, a procedural language, and a callable interface.

TPU serves as a base on which to layer other text processing applications, for example, text editors. The Extensible VAX Editor (EVE) is the editor provided with TPU. To invoke EVE, issue the following command at the DCL prompt:

```
$ EDIT/TPU USER.PAS
```

To exit from EVE, press the DO key to get the Command: prompt. If you wish to save modifications to your file, issue the EXIT command. If you do not wish to save the file or any modification to the file, issue the QUIT command.

For More Information:

For information on TPU and EVE, see the *Guide to VMS Text Processing*.

8.3 VAX Language-Sensitive Editor (LSE) and the VAX Source Code Analyzer (SCA)

The VAX Language-Sensitive Editor (LSE) and the VAX Source Code Analyzer (SCA) must be purchased separately from the VMS operating system. LSE is a text editor intended specifically for software development. SCA is an interactive tool for program analysis.

These products are closely integrated; generally, SCA can be invoked through LSE. LSE provides additional editing features that make SCA program analysis more efficient. In addition, LSE and SCA, in conjunction with the VAX Pascal compiler, provide a set of new enhancements supporting source code design and review.

In addition to text editing features, LSE provides the following software development features:

- Formatted language constructs, or templates, for most VAX programming languages, including VAX Pascal. These templates include the keywords and punctuation used in source programs, and use placeholders to indicate locations in the source code where additional text is optional or required.
- Commands to compile, review, and correct compilation errors from within the editor.
- Integration with VAX DEC/Code Management System (CMS). CMS commands can be issued from within the editor to make source file management more efficient.

SCA performs the following types of program analysis:

- Cross-referencing, which supplies information about program symbols and source files.
- Static analysis, which provides information on how subprograms, symbols, and files are related.

LSE and SCA together, in conjunction with VAX language compilers, provide the following software design features:

- Pseudocode support, which includes a new LSE placeholder for delimiting pseudocode. Pseudocode is text that describes algorithms or design decisions. This feature allows you to write source code in shorthand, returning later to fill in code details.
- Placeholder processing, in which language compilers accept LSE placeholders and pseudocode as valid program elements during compilation. This feature allows you to compile and find many errors before a program is complete.
- Comment processing, which includes design comment information in the SCA library. SCA performs cross-referencing and static analysis on this information in response to user queries.

- View support, which provides a reverse-design facility. LSE commands compress program code into overview line summaries. If you choose to edit these overview lines, the modifications you make are reflected in the program code.
- A report tool, callable through LSE, which can print views, standard design reports, and customized reports.

The following sections provide entry, exit, and language-specific information on the combined use of LSE and SCA.

For More Information:

- On LSE and SCA (*Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer*)
- On CMS (*Guide to VAX DEC/Code Management System*)

8.3.1 Preparing an SCA Library

SCA stores data generated by the VAX Pascal compiler in an SCA library. The data in an SCA library contains information about all symbols, modules, and files encountered during a specific compilation of the source. You must prepare this library before you enter LSE to invoke SCA. This preparation involves the following steps:

1. Create a VMS directory for your SCA library. For example:

```
$ CREATE/DIRECTORY PROJ:[USER.LIB1]
```

2. Initialize and set the library with the SCA CREATE LIBRARY command. For example:

```
$ SCA CREATE LIBRARY [.LIB1]
```

   If you have an existing SCA library that has been initialized, you make its contents visible to SCA by setting it with the SCA SET LIBRARY command. For example:

```
$ SCA SET LIBRARY [.EXISTING_SCA_LIBRARY]
```

3. Direct the VAX Pascal compiler to generate data analysis files by appending the /ANALYSIS_DATA qualifier to the PASCAL command. For example:

```
$ PASCAL/ANALYSIS_DATA PG1,PG2,PG3
```

This command line compiles the input files PG1.PAS, PG2.PAS, and PG3.PAS, and generates corresponding output files for each input file, with the file types .OBJ and .ANA. The VAX Pascal compiler puts these files in your current default directory.

4. Load the information in the data analysis files into your SCA library with the LOAD command. For example:

```
$ SCA LOAD PG1,PG2,PG3
```

   This command loads your library with the modules contained in the data analysis files PG1.ANA, PG2.ANA, and PG3.ANA.

5. Once the SCA library has been prepared, you can reference the SCA library by going into the SCA subsystem, or by using SCA commands from within LSE. When LSE and SCA are integrated, many additional commands and features are available.

8.3.2 Starting and Terminating an LSE or an SCA Session

To invoke LSE, issue the following command at the DCL prompt:

```
$ LSEDIT USER.PAS
```

To end an LSE session, press CTRL/Z to get the LSE> prompt. If you wish to save modifications to your file, issue the EXIT command. If you do not wish to save the file or any modification to the file, issue the QUIT command.

To invoke SCA from LSE, type the SCA command that you wish to execute at the LSE> prompt, as in the following syntax:

```
LSE>  command [parameter] [/qualifier...]
```

To invoke SCA from the DCL command line for the execution of a single command, you can use the following syntax:

```
$ SCA  command  [parameter]  [/qualifier...]
```

If you have several SCA commands to invoke, you may wish to use the SCA subsystem to enter commands, as in the following syntax:

```
$ SCA
SCA>  command [parameter] [/qualifier...]
```

Type EXIT (or press CTRL/Z) to end an SCA subsystem session and return to the DCL level.

8.3.3 Compiling from Within LSE

To compile a completed VAX Pascal program, issue the following command at the LSE prompt:

```
LSE>  COMPILE
```

To compile a VAX Pascal program that contains placeholders and design comments, and creates a data analysis file, include the following qualifiers to the previous command:

```
LSE>  COMPILE $/ANALYSIS_DATA/DESIGN
```

The /DESIGN qualifier instructs the compiler to recognize placeholders and design comments as valid program elements. To use /DESIGN, you must be running LSE Version 3.0 or higher and SCA Version 2.0 or higher. If the /ANALYSIS_DATA qualifier has also been specified, the compiler includes information on placeholders and design comments in the data analysis file.

8.3.4 Notes on VAX Pascal Support

This section describes VAX Pascal-specific information for the following LSE and SCA features:

- Programming language placeholders and tokens
- Placeholder and design comment processing

8.3.4.1 Programming Language Placeholders and Tokens

LSE accepts keywords, or tokens, for all languages with LSE support, but the specific tokens themselves are language-defined. For example, you can expand the %INCLUDE token only when using VAX Pascal.

Likewise, LSE provides placeholders, or prompt markers, for all languages with LSE support, but the specific text or choices these markers call for are language-defined. For example, you see the %{environ_name_string}% placeholder only when using VAX Pascal.

Some VAX Pascal keywords, like TYPE, VAR, IF, and FOR, can be placeholders as well as tokens. LSE supplies language constructs for these keywords when they appear on your screen as placeholders. You can also type the keywords into the buffer yourself, issue the EXPAND command, and see the same language constructs appear on your screen.

You can use the SHOW TOKEN and SHOW PLACEHOLDER commands to display a list of all VAX Pascal tokens and placeholders, or a particular token or placeholder. For example:

```
LSE> SHOW TOKEN IF                    {lists the token IF}
LSE> SHOW TOKEN                       {lists all tokens  }
```

To copy the listed information into a separate file, first issue the appropriate SHOW command to put the list into the $SHOW buffer. Then issue the following command:

```
LSE> GOTO BUFFER $SHOW
LSE> WRITE filename.filetype
```

8.3.4.2 Placeholder and Design Comment Processing

While all languages with LSE support provide placeholder processing, each language defines specific contexts in which placeholders can be accepted as valid program code. VAX Pascal defines contexts for declaration section placeholders and executable section placeholders. Table 8–1 lists the valid contexts within a VAX Pascal declaration section.

Table 8–1: Placeholders Within the Declaration Section

Can Replace	Cannot Replace
PROGRAM or MODULE identifier	Directive
Program parameter	Attribute
Identifier	Declaration-begin reserved word
Data type	Complete declaration
Value	
Complete variant within the variant part of record	

Table 8–2 lists valid contexts within a VAX Pascal executable section.

Table 8–2: Placeholders Within the Executable Section

Can Replace	Cannot Replace
Statement	LABEL identifier
Variable	TO | DOWNTO within a FOR statement
Expression	
Case label	
Complete case expression	
Iteration variable within a FOR statement	

VAX Pascal support for placeholder and design comment processing includes the following language-specific stipulations:

- Pseudocode placeholders are designated with double left- and right-angle brackets (<< >>) or the 8 bit format (« »).
- The compiler produces an empty object file when it encounters pseudocode or LSE placeholders within a source program.
- Comment processing is limited to the declaration section.

8.3.5 LSE and SCA Examples

Example 8–1 shows how you can use LSE tokens and placeholders to create a FOR statement within a VAX Pascal program. The callout numbers identify the steps in this process, which are detailed in the notes appearing after the example.

Example 8–1: Using LSE to Create a FOR Statement

```
❶ BEGIN
  %[statement_list]%...
  END.
     .
     .
     .
❷ BEGIN
   FOR %{control_var}% %{iteration_clause}% DO
       %{statement}%;
  %[statement_list]%...
  END.
     .
     .
     .
  BEGIN
  FOR INDEX := 1 TO MAX DO
❸     %{statement}%;
  %[statement_list]%...
  END.
     .
     .
     .
  BEGIN
  FOR INDEX := 1 TO MAX DO
❹     %{variable | func_id}% := %{value_expr}%;
  %[statement_list]%...
  END.
     .
     .
     .
  BEGIN
❺ FOR INDEX := 1 TO MAX DO
      ARR[INDEX] := 0;
  %[statement_list]%...
  END.
```

❶ As you begin the executable section of your program, the cursor rests on the placeholder %[statement_list]%. Type the token FOR over this placeholder and expand FOR.

❷ LSE provides the FOR statement template. Select a FOR variable option from the menu. Expand the %{iteration_clause}% placeholder and expand the %{statement}% placeholder.

❸ LSE displays a menu, from which you can select the %{simple_statement}% option. A further menu appears, from which you select the ASSIGNMENT statement option.

❹ LSE provides the assignment statement template. Type an appropriate identifer or value expression over each placeholder.

❺ The completed FOR statement appears in your buffer.

Example 8–2 shows some contexts in which LSE placeholders and design comments might appear in the design of a VAX Pascal program. Placeholder contexts are self-explanatory; the callout numbers identify types of comments, which are detailed in the notes following the example.

Example 8–2: Using LSE Comments in Program Design

```
PROGRAM Semester_Grades ( input,output ) ;

❶
{ Author : P. Knox                                                  }
{ Creation Date : 03/03/89                                          }

{ Functional Description :
  This program calculates the numerical semester grade
  and determines the corresponding alphabetic grade
  for each student in a class.                                      }

{ Non-local References : None                                       }

{ Included Files : None                                             }

{ Keywords :
  Grade array procedures, semester grade file                       }

CONST
    «number of students» = %{compile_time_exp}%;
    «number of semester grades» = %{compile_time_exp}%;

TYPE
    «grade array» = ARRAY [ 1..«number of semester grades»] OF
                      «integer or real? determine later»;
    «grade range» = 0 .. 100;

❷ VAR
    A_Grade_Array  : «grade array» ;  { var for array of grades  }
    A_Grade        : «grade range» ;  { var for individual grade }
    Total          : «grade range» ;  { var for semester grade   }
    %{variable}% : INTEGER;           { control var, FOR loops   }

PROCEDURE «compute semester grade»
    ( VAR Grd_Arry : «grade array» ;
      VAR Sum : «grade range» ) ;

❸
{ Parameters :
     Grd_Arry : value parameter, array of semester grades for one
                student:

     Sum      : variable parameter, returns the semester grade }
```

(continued on next page)

Example 8–2 (Cont.): Using LSE Comments in Program Design

```
    BEGIN
       FOR %{control_var}% %{iteration_clause}% DO
           «sum the grades in Grd_Arry»
       «Sum gets Sum divided by number of grades in array»
       «write student's semester grade»
     END;

PROCEDURE «assign letter grade»
    ( Semester_Grade : «grade range» ) ;

    BEGIN
       CASE %{case_selector}% OF
             %{case_labels}%... : %{statement}%;
             %{case_labels}%... : %{statement}%;
             %{case_labels}%... : %{statement}%;
             %{case_labels}%... : %{statement}%
             OTHERWISE %{statement_list}%...
       END
    END;

BEGIN
FOR «number of students» DO
    BEGIN
    «enter semester grades for student» ;
    FOR %{control_var}% := %{value_expr}% TO
                           «number of semester grades» DO
       BEGIN
             «read grades and load into array»
       END;
    Total := 0;
    «compute routine» ( Grade_Array, Total );
    «assign_letter routine» ( Total )
    END
END.
```

❶ These comments, which could be placed at the beginning of the program, are tagged comments. The comment begins with a predefined term called a tag. The tag is followed by a tag terminator symbol (:) and free text.

❷ These comments, which do not contain tags, are remark comments. A remark comment consists of free text.

❸ This structured comment contains both a tag (the identifier "parameters") and subtags (the identifiers "Grd_Arry" and "Sum"). A tag terminator and a blank comment line separate the two subtags from each other.

8.4 VAX Common Data Dictionary (CDD)

The VAX Common Data Dictionary (CDD) must be purchased separately from the VMS operating system. The CDD allows language-independent structure declarations that can be shared by many VMS layered products. VAX Pascal support of the CDD allows VAX Pascal programmers to share common record and data definitions with other VAX languages and VAX data management products.

A system manager or data administrator creates the CDD's directory hierarchies, history lists, and access control lists with the Dictionary Management Utility (DMU). Once record paths are established, data definitions can be entered into and extracted from the CDD.

To enter data definitions into the CDD, you first create CDD source files written in the Common Data Dictionary Language (CDDL). The CDDL compiler converts the definitions to an internal form—making them independent of the language used to access them.

To extract data definitions from the CDD, include the %DICTIONARY directive in your VAX Pascal source program. If the data attributes of the data definitions are consistent with VAX Pascal requirements, the data definitions are included in the VAX Pascal program during compilation.

For More Information:

- On CDD (*VAX Common Data Dictionary Utilities Reference Manual*)
- On CDDL (*VAX Common Data Dictionary Data Definition Language Reference Manual*)
- On CDDL data types (*VAX Common Data Dictionary User's Guide* and *VAX Common Data Dictionary Data Definition Language Reference Manual*)
- On the VAX Pascal %DICTIONARY directive (*VAX Pascal Reference Manual*)

8.4.1 Accessing the CDD from VAX Pascal Source Programs

The %DICTIONARY directive incorporates VAX Common Data Dictionary data definitions into the current VAX Pascal source file during compilation.

This directive can appear only in the TYPE section of a VAX Pascal program, not in the executable section. For example:

```
PROGRAM SAMPLE1;

TYPE
   %DICTIONARY 'Pascal_SALESMAN_RECORD/LIST'
   .
   .
   .
```

A /LIST option in the %DICTIONARY directive (or the /SHOW= DICTIONARY qualifier on the Pascal command line) includes the translated record in the program's listing. For example:

```
TYPE
   %DICTIONARY 'PASCAL_SALESMAN_RECORD/LIST'
   { CDD Path Name => PASCAL_SALESMAN_RECORD }

   PAYROLL_RECORD = PACKED RECORD
      SALESMAN    :  PACKED RECORD
      NAME        :  PACKED ARRAY [1..30] OF CHAR;
      ADDRESS     :  PACKED ARRAY [1..40] OF CHAR;
      SALESMAN_ID :  [BYTE(5)] RECORD END; { numeric string, unsigned }
      END; { record salesman }
   END; { record payroll_record }
```

The option (/LIST or /NOLIST) overrides the qualifier (/SHOW= NODICTIONARY or /SHOW=DICTIONARY).

8.4.2 Equivalent VAX Pascal and CDDL Data Types

The CDD supports some data types that are not native to VAX Pascal. If a data definition contains a field declared with an unsupported data type, VAX Pascal replaces the field with one declared as a [BYTE(n)] RECORD END, where n is the appropriate length in bytes. By making the data addressable in this way, you are able to manipulate the data either by passing it to external routines as variables or by using the VAX Pascal type casting capabilities to perform an assignment.

However, because these empty records do not have fields, the size of the record is 0 bits. They should not be used in expressions or passed to formal value parameters. Recall that a size attribute used on a type definition has no effect on fetches. When fetching from these records, the compiler will fetch the actual size of the record, 0 bits.

Table 8–3 summarizes the mapping between CDDL data types and the corresponding VAX Pascal data types.

NOTE

D_floating and G_floating data types cannot be mixed between routines and compilation units; however, both types cannot be handled in the same expression. Not all VAX processors support the G_floating and H_floating types.

Table 8–3: Equivalent CDDL and VAX Pascal Data Types

CDDL Data Type	VAX Pascal Data Type
Unspecified	[BYTE(n)] RECORD END
Byte logical	[BYTE] 0..255
Word logical	[WORD] 0..65535
Longword logical	UNSIGNED
Quadword logical	[BYTE(8)] RECORD END
Octaword logical	[BYTE(16)] RECORD END
Byte integer	[BYTE] –128..127
Word integer	[WORD] –32768..32767
Longword integer	INTEGER
Quadword integer	[BYTE(8)] RECORD END
Octaword integer	[BYTE(16)] RECORD END
F_floating	SINGLE
D_floating	DOUBLE (/NOG_FLOATING)
G_floating	DOUBLE (/G_FLOATING)
H_floating	QUADRUPLE
F_floating complex	[BYTE (8)] RECORD END
D_floating complex	[BYTE(16)] RECORD END
G_floating complex	[BYTE(16)] RECORD END
H_floating complex	[BYTE(32)] RECORD END
Text	PACKED ARRAY [l..u] OF CHAR
Varying text	VARYING [u] OF CHAR
Numeric string, unsigned	[BYTE(n)] RECORD END

(continued on next page)

Table 8–3 (Cont.): Equivalent CDDL and VAX Pascal Data Types

CDDL Data Type	VAX Pascal Data Type
Numeric string, left separate	[BYTE(n)] RECORD END
Numeric string, left overpunch	[BYTE(n)] RECORD END
Numeric string, right separate	[BYTE(n)] RECORD END
Numeric string, right overpunch	[BYTE(n)] RECORD END
Numeric string, zoned sign	[BYTE(n)] RECORD END
Bit	[BIT(n)] 0..((2^n)–1) or [BIT(32)]UNSIGNED or [BIT(N)] RECORD END or ignored
Bit unaligned	[BIT(n), POS(x)] 0..((2^n)n–1) or [BIT(32), POS(x)] UNSIGNED or [BIT(n), POS(x)] RECORD END or ignored
Date and time	[BYTE(n)] RECORD END
Date	[BYTE(n)] RECORD END
Virtual field	ignored
Varying string	VARYING [u] OF CHAR
Overlay	variant record
Pointer	pointer type

8.4.3 CDD Example

In Example 8–3, the %DICTIONARY directive is used to access the CDD record definition Mail_Order_Info. With this definition, the VAX Pascal program Show_Keys performs ISAM file manipulation on an existing indexed file, Customers.Dat. Assume that Customers.Dat has the primary key Order_Num and alternate keys Zip_Code and Item_Num.

NOTE

The CDD has no equivalent for the VAX Pascal KEY attribute, which is required to create new indexed files. CDD data definitions can be used to open existing indexed files (as in this example) but not new indexed files.

Example 8–3: Using %DICTIONARY to Access a CDD Record Definition

```
Program Show_Keys(OUTPUT);

TYPE
    %DICTIONARY 'Mail_Order_Info/LIST'

VAR
    Old_Customer_File   : FILE OF Mail_Order;
    Order_Rec           : Mail_Order;
    Continue            : BOOLEAN;

BEGIN
OPEN( File_Variable := Old_Customer_File,
      File_Name     := 'Customers.Dat',
      History       := OLD,
      Organization  := Indexed,
      Access_Method := Keyed );

FINDK(Old_Customer_File, 1, '10000', NXTEQL);
Continue := TRUE;
WHILE Continue and NOT UFB(Old_Customer_File) DO
    BEGIN
    READ(Old_Customer_File, Order_Rec);
    IF Order_Rec.Zip_Code < '5000'
    THEN
        WRITELN('Order number', Order_Rec.Order_Num, 'has zip code',
        Order_Rec.Zip_Code)
    ELSE
        Continue := False;
    END;
END.
```

During the compilation of Show_Keys, the record definition Mail_Order_Info is extracted from the CDD. Show_Keys prints the order number and zip code of each file component that has a zip code greater than or equal to 1000 but less than 5000.

Appendix A

Environment-Specific Implementation Features

This appendix lists implementation features that are specific to the VMS operating system or to the VAX architecture. Implementation features are those features that the ISO standard for Pascal allows to be defined by a particular implementation or to be dependent on an implementation. For more information on implementation features that are not system- or architecture-specific, see the *VAX Pascal Reference Manual*.

A.1 Implementation-Defined Features

The range of real number values represented by the type REAL

Treatment: See Chapter 2.

The point at which the REWRITE, PUT, RESET, and GET procedures are performed on a file

Treatment: Performed immediately unless the file is a terminal file, in which case delayed device access occurs (see the *VAX Pascal User Manual*).

The accuracy to which the results of real-number operations are calculated

Treatment: See Chapter 2.

The number of digits used to represent the exponent of a floating-point number

Treatment:

F_floating or D_floating	2
G_floating	3
H_floating	4

The binding of a file variable whose name is listed in the program heading

Treatment: The file name (unless it is INPUT or OUTPUT) is equated to a logical name if a translation for the file name exists. If there is no corresponding translation, the file type DAT is appended to the name listed in the heading, as in INFILE.DAT. If the file name is INPUT, the file is equated to PAS$INPUT, if PAS$INPUT is defined; otherwise, the file is equated to SYS$INPUT. Similarly, if the file name is OUTPUT, the file is equated to PAS$OUTPUT, if PAS$OUTPUT is defined; otherwise, the file is equated to SYS$OUTPUT.

Appendix B

Diagnostic Messages

This appendix summarizes the error messages that can be generated by a VAX Pascal program at compile time and at run time.

B.1 Compiler Diagnostics

The VAX Pascal compiler reports compile-time diagnostics in the source listing (if one is being generated) and summarizes them on the terminal (in interactive mode) or in the batch log file (in batch mode). Compile-time diagnostics are preceded by the following:

```
          I-
%PASCAL-  W-
          E-
          F-
```

- I indicates an informational message that flags VAX extensions to the Pascal standard, identifies unused or possibly uninitialized variables, or provides additional information about a more severe error.
- W indicates a warning that flags an error or construct that may cause unexpected results, but that does not prevent the program from linking and executing.
- E indicates an error that prevents generation of machine code; instead, the compiler produces an empty object module indicating that E-level messages were detected in the source program.
- F indicates a fatal error.

If the source program contains either E- or F-level messages, the errors must be corrected before the program can be linked and executed.

All diagnostic messages contain a brief explanation of the event that caused the error. This section lists compile-time diagnostic messages in alphabetical order, including their severity codes and explanatory message text. Where the message text is not self-explanatory, additional explanation follows. Portions of the message text enclosed in quotation marks are items that the compiler substitutes with the name of a data object when it generates the message.

ABSALIGNCON, Absolute address / alignment conflict

Error: The address specified by the AT attribute does not have the number of low-order bits implied by the specified alignment attribute.

ACCMETHCON, Specified ACCESS_METHOD conflicts with file's record organization

Warning: You cannot specify ACCESS_METHOD:=DIRECT for a file that has indexed organization or sequential organization and variable-length records. You cannot specify ACCESS_METHOD:=KEYED for a file with sequential or relative organization.

ACTHASNOFRML, Actual parameter has no corresponding formal parameter

Error: The number of actual parameters specified in a routine call exceeds the number of formal parameters in the routine's declaration, and the last formal parameter does not have the LIST attribute.

ACTMULTPL, Actual parameter specified more than once

Error: Each formal parameter (except one with the LIST attribute) can have only one corresponding actual parameter.

ACTPASCNVTMP, Conversion: actual passed is resulting temporary
ACTPASRDTMP, Formal requires read access: actual parameter is resulting temporary
ACTPASSIZTMP, Size mismatch: actual passed is resulting temporary
ACTPASWRTMP, Formal requires write access: actual parameter is resulting temporary

Warning: A temporary variable is created if an actual parameter does not have the size, type, and accessibility properties required by the corresponding foreign formal parameter.

ACTPRMORD, Actual parameter must be ordinal

Error: The actual parameter that specifies the starting index of an array for the PACK or UNPACK procedure must have an ordinal type.

ADDIWRDALIGN, ADD_INTERLOCKED requires variable with at least word alignment

ADDIWRDSIZE, ADD_INTERLOCKED requires 16-bit variable

Error: These restrictions are imposed by the VAX ADAWI instruction.

ADDRESSVAR, "parameter name" is a VAR parameter, ADDRESS is illegal

Warning: You should not use the ADDRESS function on a nonvolatile variable or component or on a formal VAR parameter.

ADISCABSENT, Formal discriminant "discriminant name" has no corresponding actual discriminant

Error: An actual discriminant must be specified for every formal discriminant in a schema type definition.

ADISCHASNOFRML, Actual discriminant has no corresponding formal discriminant

Error: The number of actual discriminants specified is greater than the number of formal discriminants defined in the schema type definition.

ALIATRTYPCON, Alignment attribute / type conflict

ALIGNAUTO, Alignment greater than 2 conflicts with automatic allocation

Error: The VAX hardware aligns the stack on a longword boundary; therefore, you cannot specify a greater alignment for automatically allocated variables.

ALIGNFNCRES, Alignment greater than 2 not allowed on function result

Error: The use of an attribute on a routine conflicts with the requirements of the object's type.

ALIGNINT, ALIGNED expression must be INTEGER value in range 0..9

Error.

ALIGNVALPRM, Alignment greater than 2 not allowed on value parameter

Error: The use of an attribute on a parameter conflicts with the requirements of the object's type.

ALLPRMSAM, All parameters to 'MIN' or 'MAX' must have the same type

Error.

APARMACTDEF, Anonymous parameter "parameter number" has neither actual nor default

Error: If the declaration of a routine failed to specify a name for a formal parameter, a call to the routine will result in this error message. Note that the routine declaration would also cause an error to be reported.

ARITHOPNDREQ, Arithmetic operand(s) required

Error.

ARRCNTPCK, Array cannot be PACKED

Error: At least one parameter to the PACK or UNPACK procedure must be unpacked.

ARRHAVSIZ, "routine name" requires that ARRAY component have compile-time known size

Error: You cannot use the PACK and UNPACK procedures to pack or unpack one multidimensional conformant array into another. The component type of the dimension being copied must have a compile-time known size; that is, it must have some type other than a conformant schema.

ARRMSTPCK, Array must be PACKED

Error: At least one parameter to the PACK or UNPACK procedure must be of type PACKED.

ARRNOTSTR, Array type is not a string type

Error: You cannot write a value to a text file (using WRITE or WRITELN) or to a VARYING string (using WRITEV) if there is no textual representation for the type. Similarly, you cannot read a value from a text file (using READ or READLN) or from a VARYING string (using READV) if there is no textual representation for the type. The only legal array, therefore, is PACKED ARRAY [1..n] OF CHAR.

ASYREQASY, ASYNCHRONOUS "calling routine" requires that "called routine" also be ASYNCHRONOUS

Warning.

ASYREQVOL, ASYNCHRONOUS "routine name" requires that "variable name" be VOLATILE

Warning: A variable referred to in a nested asynchronous routine must have the VOLATILE attribute.

ATINTUNS, AT address must be an INTEGER or UNSIGNED value

Error.

ATREXTERN, "attribute name" attribute allowed only on external routines

Error: The LIST and CLASS_S attributes can be specified only with the declarations of external routines.

ATTRCONCMDLNE, Attribute contradicts command line qualifier

Error: The double-precision attribute specified contradicts the /G_FLOATING or /NOG_FLOATING qualifier specified with the PASCAL command.

ATTRCONFLICT, Attribute conflict: "attribute name"

Information: This message can appear as additional information on other error messages.

ATTRONTYP, Descriptor class attribute not allowed on this type

Error: The use of the descriptor class attribute on the variable, parameter, or routine conflicts with the requirements of the object's type.

AUTOGTRMAXINT, Allocation of "variable name" causes automatic storage to exceed MAXINT bits

Error: The VAX implementation restricts automatic storage to a size of 2,147,483,647 bits.

BADANAORG, Analysis data file "file name" is not on a random access device

Fatal.

BADENVORG, Environment file "file name" is not on a random access device

Fatal.

BADSETCMP, < and > not permitted in set comparisons

Error.

BINOCTHEX, Expecting BIN, OCT, or HEX

Error: You must supply BIN, OCT, or HEX as a variable modifier when reading the variable on a nondecimal basis.

BLKNOTFND, "routine" block "routine name" declared FORWARD in "block name" is missing

Error.

BLKTOODEEP, Routine blocks nested too deeply

Error: You cannot nest more than 31 routine blocks.

BNDACTDIFF, Actual's array bounds differ from those of other parameters in same section

Error: All actual parameters passed to a formal parameter section whose type is a conformant schema must have identical bounds and be structurally compatible.

BNDCNFRUN, Bounds of conformant ARRAY "array name" not known until run-time

Error: You cannot use the UPPER and LOWER functions on a dynamic array parameter in a compile-time constant expression.

BNDSUBORD, Bound expressions in a subrange type must be ordinal

Error: The expressions that designate the upper and lower limits of a subrange must be of an ordinal type.

BOOLOPREQ, BOOLEAN operand(s) required

Error: The operation being performed requires operands of type BOOLEAN. Such operations include the AND, OR, and NOT operators and the SET_INTERLOCKED and CLEAR_INTERLOCKED functions.

BOOSETREQ, BOOLEAN or SET operand(s) required

Error.

BYTEALIGN, Type larger than 32 bits can be positioned only on a byte boundary

Error: See Chapter 2 for information on the types that are allocated more than 32 bits.

CALLFUNC, Function "function name" called as procedure, function value discarded

Warning.

CARCONMNGLS, CARRIAGE_CONTROL parameter is meaningless given file's type

Warning: The carriage control parameter is usually meaningful only for files of type TEXT and VARYING OF CHAR.

CASLABEXPR, Case label and case selector expressions are not compatible

Error: All case labels in a CASE statement must be compatible with the expression specified as the case selector.

CASORDRELPTR, Compile-time cast allowed only between ordinal, real, and pointer types

CASSELORD, Case selector expression must be an ordinal type

Error.

CASSRCSIZ, Source type of a cast must have a size known at compile-time
CASTARSIZ, Target type of a cast must have a size known at compile-time

Error: A variable being cast by the type cast operator cannot be a conformant array or a conformant VARYING parameter. An expression being cast cannot be a conformant array parameter, a conformant VARYING parameter, or a VARYING OF CHAR expression. The target type of the cast cannot be VARYING OF CHAR.

CDDABORT, %DICTIONARY processing of CDD record definition aborted

Error: The VAX Pascal compiler is unable to process the CDD record description. See the accompanying CDD messages for more information.

CDDBADDIR, %DICTIONARY directive not allowed in deepest %INCLUDE, ignored

Error: A program cannot use the %DICTIONARY directive in the fifth nested %INCLUDE level. The compiler ignores all %DICTIONARY directives in the fifth nested %INCLUDE level.

CDDBADPTR, invalid pointer was specified in CDD record description

Warning: The CDD pointer data type refers to a CDD path name that cannot be extracted, and is replaced by ^INTEGER.

CDDBIT, Ignoring bit field in CDD record description

Information: The VAX Pascal compiler cannot translate a CDD bit data type that is not aligned on a byte boundary and whose size is greater than 32 bits.

CDDBLNKZERO, Ignoring blank when zero attribute specified in CDD record description

Information: The VAX Pascal compiler does not support the CDD BLANK WHEN ZERO clause.

CDDCOLMAJOR, CDD description specifies a column-major array

Error: The VAX Pascal compiler supports only row-major arrays. Change the CDD description to specify a row-major array.

CDDDEPITEM, Ignoring depends item attribute specified in CDD record description

Information: The VAX Pascal compiler does not support the CDD DEPENDING ON ITEM attribute.

CDDDFLOAT, D_Floating CDD datatype was specified when compiling with G_FLOATING

Warning: The CDD record description contains a D_floating data type while compiling with G_floating enabled. It is replaced with [BYTE(8)] RECORD END.

CDDFLDVAR, CDD record description contains field(s) after CDD variant clause

Error: The CDD record description contains fields after the CDD variant clause. Because VAX Pascal translates a CDD variant clause into a Pascal variant clause, and a Pascal variant clause must be the last field in a record type definition, the fields following the CDD variant clause are illegal.

CDDGFLOAT, G_Floating CDD datatype was specified when compiling with NOG_FLOATING

Warning: The CDD record description contains a G_floating data type while compiling with D_floating enabled. It is replaced with [BYTE(8)] RECORD END.

CDDILLARR, Aligned array elements can not be represented, replacing with [BIT(n)] RECORD END

Information: The VAX Pascal compiler does not support CDD record descriptions that specify an array whose array elements are aligned on a boundary greater than the size needed to represent the data type. It is replaced with [BIT(n)] RECORD END, where n is the appropriate length in bits.

CDDINITVAL, Ignoring specified initial value specified in CDD record description

Information: The VAX Pascal compiler does not support the CDD INITIAL VALUE clause.

CDDMINOCC, Ignoring minimum occurs attribute specified in CDD record description

Information: The VAX Pascal compiler does not support the CDD MINIMUM OCCURS attribute.

CDDONLYTYP, %DICTIONARY may only appear in a TYPE definition part

Error: The %DICTIONARY directive is allowed only in the TYPE section of a program.

CDDRGHTJUST, Ignoring right justified attribute specified in CDD record description

Information: The VAX Pascal compiler does not support the CDD JUSTIFIED RIGHT clause.

CDDSCALE, Ignoring scaled attribute specified in CDD record description

Information: The VAX Pascal compiler does not support the CDD scaled data types.

CDDSRCTYPE, Ignoring source type attribute specified in CDD record description

Information: The VAX Pascal compiler does not support the CDD source type attribute.

CDDTAGDEEP, CDD description nested variants too deep

Error: A CDD record description may not include more than 15 levels of CDD variants. The compiler ignores variants beyond the fifteenth level.

CDDTAGVAR, Ignoring tag variable and any tag values specified in CDD record description

Information: The VAX Pascal compiler does not fully support CDD VARIANTS OF field description statement. The specified tag variable and any tag values are ignored.

CDDTOODEEP, CDD description nested too deep

Error: Attributes for the CDD record description exceed implementation's limit for record complexity. Modify the CDD description to reduce the level of nesting in the record description.

CDDTRUNCREF, Reference string which exceeds 255 characters has been truncated

Information: The VAX Pascal compiler does not support reference strings greater than 255 characters.

CDDUNSTYP, Unsupported CDD datatype "standard data type name"

Information: The CDD record description for an item has attempted to use a data type that is not supported by VAX Pascal. The VAX Pascal compiler makes the data type accessible by declaring it as [BYTE(n)] RECORD END where n is the appropriate length in bytes. Change the data type to one that is supported by VAX Pascal or manipulate the contents of the field by passing it to external routines as variables or by using the VAX Pascal type casting capabilities to perform an assignment.

CLSCNFVAL, CLASS_S is only valid with conformant strings

Error: When the CLASS_S attribute is used in the declaration of an internal routine, it can be used only on a conformant PACKED ARRAY OF CHAR. The conformant variable must also be passed by value semantics.

CLSNOTALLW, "descriptor class name" not allowed on a parameter of this type

Error: Descriptor class attributes are not allowed on formal parameters defined with either an immediate or a reference passing mechanism.

CMTBEFEOF, Comment not terminated before end of input

Error.

CNFCANTCNF, Component of PACKED conformant parameter cannot be conformant

Error.

CNFREQNCA, Conformants of this parameter type require CLASS_NCA

Error: The conformant parameter cannot be described with the default CLASS_A descriptor. Add the CLASS_NCA attribute to the parameter declaration.

CNSTRNOTALL, Nonstandard constructors are not allowed on nonstatic types

Error: You can write constructors for nonstatic types using the standard style of constructor.

CNSTRONZERO, Record constructors only allow OTHERWISE ZERO

Error.

CNTBEARRCMP, Not allowed on an array component
CNTBEARRIDX, Not allowed on an array index
CNTBECAST, Not allowed on a cast
CNTBECNFCMP, Not allowed on a conformant array component
CNTBECNFIDX, Not allowed on a conformant array index
CNTBECNFVRY, Not allowed on a conformant varying component
CNTBECOMP, Not allowed on a compilation unit
CNTBECONST, Not allowed on a CONST definition part

CNTBEDEFDECL, Not allowed on any definition or declaration part
CNTBEDESPARM, Not allowed on a %DESCR foreign mechanism parameter
CNTBEEXESEC, Not allowed on an executable section
CNTBEFILCMP, Not allowed on a file component
CNTBEFORMAL, Not allowed on a formal discriminant
CNTBEFUNC, Not allowed on a function result
CNTBEIMMPARM, Not allowed on a parameter passed by an immediate passing mechanism
CNTBELABEL, Not allowed on a LABEL declaration part
CNTBEPCKCNF, Not allowed on a PACKED conformant array component

CNTBEPTRBAS, Not allowed on a pointer base
CNTBERECFLD, Not allowed on a record field
CNTBEREFPARM, Not allowed on a parameter passed by a reference passing mechanism
CNTBERTNDECL, Not allowed on a routine declaration
CNTBERTNPARM, Not allowed on a routine parameter
CNTBESCHEMA, Not allowed on a nonstatic type
CNTBESETRNG, Not allowed on a set range
CNTBESTDPARM, Not allowed on a %STDESCR foreign mechanism parameter
CNTBETAGFLD, Not allowed on a variant tag field

CNTBETAGTYP, Not allowed on a variant tag type
CNTBETO, Not allowed on TO BEGIN/END DO
CNTBETYPDEF, Not allowed on a type definition
CNTBETYPE, Not allowed on a TYPE definition part
CNTBEVALPARM, Not allowed on a value parameter
CNTBEVALUE, Not allowed on a VALUE initialization part
CNTBEVALVAR, Not allowed on a VALUE variable
CNTBEVAR, Not allowed on a VAR declaration part
CNTBEVARBLE, Not allowed on a variable

CNTBEVARPARM, Not allowed on a VAR parameter
CNTBEVRYCMP, Not allowed on a varying component

Information: These messages can appear as additional information on other error messages.

COMCONFLICT, COMMON "block name" conflicts with another COMMON or PSECT of same name

Error: You can allocate only one variable in a particular common block, and the name of the common block cannot be the same as the names of other common blocks or program sections used by your program.

CSTRBADTYP, Constructor: only ARRAY, RECORD, or SET type
CSTRCOMISS, Constructor: component(s) missing
CSTRNOVRNT, Constructor: no matching variant
CSTRREFAARR, Repetition factor allowed only in ARRAY constructors
CSTRREFAINT, Repetition factor must be INTEGER
CSTRREFALRG, Repetition factor too large
CSTRREFANEG, Repetition factor cannot be negative
CSTRTOOMANY, Constructor: too many components

Error: You can write constructors only for data items of an ARRAY type. You must specify one and only one value in the constructor for each component of the type. In an array constructor, you cannot use a negative integer value as a repetition factor to specify values for consecutive components.

CTESTRSIZ, Compile-time strings must be less than 8192 characters

Error.

CTGARRDESC, Contiguous array descriptor cannot describe size/alignment properties

Information: Conformant array parameters, dynamic array parameters, and %DESCR array parameters all use the contiguous array descriptor mechanism in the VAX Procedure Calling Standard. Size and alignment attributes are prohibited on such arrays, as these attributes can create noncontiguous allocation. This message can appear as additional information in other error messages.

DEBUGOPT, /NOOPTIMIZE is recommended with /DEBUG

Information: Unexpected results may be seen when debugging an optimized program. To prevent conflicts between optimization and debugging, you should compile your program with /NOOPTIMIZE until it is thoroughly debugged. Then you can recompile the program with optimization enabled to produce more efficient code.

DECLORDER, Declarations are out of order

Error: The TO BEGIN DO and TO END DO declarations in a module must appear at the end of the module and may not be reordered.

DEFRTNPARM, Default parameter syntax not allowed on routine parameters

DEFVARPARM, Default parameter syntax not allowed on VAR parameters

Error.

DESCOMABORT, Further processing of /DESIGN=COMMENTS has been aborted

Error: An error has occurred that prohibits further comment processing.

DESCOMERR, An error has occurred while processing design information

Error.

DESCOMSEVERR, An internal error has occurred while processing /DESIGN=COMMENTS - please submit an SPR

Error: A fatal error has occurred during comment processing. Please submit an SPR including sufficient information to reproduce the program, including the version numbers of VAX LSE and the VAX Pascal compiler.

DESCTYPCON, Descriptor class / type conflict

Error: The descriptor class for parameter passing conflicts with the parameter's type. Refer to Section 3.3.3 for legal descriptor class/type combinations.

DESIGNTOOOLD, The comment processing routines are too old for the compiler

Error: The support routines for the /DESIGN=COMMENT qualifier are obsolete. Contact your system manager.

DIRCONVISIB, Directive contradicts visibility attribute

Error: The EXTERN, EXTERNAL, and FORTRAN directives conflict directly with the LOCAL and GLOBAL attributes.

DISCLIMIT, Limit of 255 discriminants exceeded

Error.

DISNOTORD, Discriminant type must be an ordinal type

Error: The formal discriminant in a schema type definition must be an ordinal type.

DONTPACKVAR, "routine name" is illegal, variable can never appear in a packed context

Error: You cannot call the BITSIZE and BITNEXT functions for conformant parameters.

DUPLALIGN, Alignment already specified
DUPLALLOC, Allocation already specified
DUPLATTR, Attribute already specified
DUPLCLASS, Descriptor class already specified
DUPLDOUBLE, Double precision already specified

Error: Only one member of a particular attribute class can appear in the same attribute list.

DUPLFIN, TO END DO already specified
DUPLINIT, TO BEGIN DO already specified

Error: Only one TO BEGIN DO and one TO END DO section can appear in the same module.

DUPLGBLNAM, Duplicated global name

Warning: The GLOBAL attribute cannot appear on more than one variable or routine with the same name.

DUPLMECH, Passing mechanism already specified
DUPLOPT, Optimization already specified
DUPLSIZE, Size already specified
DUPLVISIB, Visibility already specified

Error: Only one member of a particular attribute class can appear in the same attribute list.

DUPTYPALI, Alignment already specified by type identifier "type name"
DUPTYPALL, Allocation already specified by type identifier "type name"
DUPTYPATR, Attribute already specified by type identifier "type name"
DUPTYPDES, Descriptor class already specified by type identifier "type name"
DUPTYPSIZ, Size already specified by type identifier "type name"
DUPTYPVIS, Visibility already specified by the type identifier "type name"

Error: An attribute specified for an object was already specified in the definition of the object's type.

ELEOUTRNG, Element out of range

Error: A value specified in a set constructor used as a compile-time constant expression does not fall within the subrange defined as the set's base type.

EMPTYCASE, Empty case body

Error: You failed to specify any case labels and corresponding statements in the body of a CASE statement.

ENVERROR, Environment resulted from a compilation with Errors

Error: When a program inherits an environment file that compiled with errors, unexpected results may occur during the program's compilation. The environment file inherited by the program compiled with errors. Unexpected results may occur in the program now being compiled.

ENVFATAL, Environment resulted from a compilation with Fatal Errors

Error: The environment file inherited by the program compiled with fatal errors. Unexpected results may occur in the program now being compiled.

ENVOLDVER, Environment was created by a VAX Pascal V2 compiler, please recompile

Warning: The environment file inherited by the program was created by a VAX Pascal Version 2.0 compiler. You should regenerate the environment file with the VAX Pascal Version 3.0 compiler.

ENVWARN, Environment resulted from a compilation with Warnings

Warning: The environment file inherited by the program compiled with warnings. Unexpected results may occur in the program now being compiled.

ERREALCNST, Error in real constant: digit expected

Error.

ERRNONPOS, ERROR parameter can be specified only with nonpositional syntax

Error.

ERRORLIMIT, Error Limit = "current error limit", source analysis terminated

Fatal: The error limit specified for the program's compilation was exceeded; the compiler was unable to continue processing the program. By default, the error limit is set at 30, but you can use the /ERROR_LIMIT qualifier at compile time to change it.

ESTBASYNCH, ESTABLISH requires that "routine name" be ASYNCHRONOUS

Warning.

EXPLCONVREQ, Explicit conversion to lower type required

Error: An expression of a higher-ranked type cannot be assigned to a variable of a lower-ranked type; you must first convert the higher-ranked expression by using DBLE, SNGL, TRUNC, ROUND, UTRUNC, or UROUND, as appropriate.

EXPRARITH, Expression must be arithmetic

Error: An expression whose type is not arithmetic cannot be assigned to a variable of a real type.

EXPRARRIDX, Expression is incompatible with unpacked array's index type

Error: The index type of the unpacked array is not compatible with the index type of either the PACK or UNPACK procedure it was passed to.

EXPRCOMTAG, Expression is not compatible with tag type

Error: A case label specified for a NEW, DISPOSE, or SIZE routine must be assignment compatible with the tag type of the variant.

EXPRNOTSET, Expression is not a SET type

Error: The compiler encountered an expression of some type other than SET in a call to the CARD function.

EXTRNALLOC, Allocation attribute conflicts with EXTERNAL visibility

Error: The storage for an external variable or routine is not allocated by the current compilation; therefore, the specification of an allocation attribute is meaningless.

EXTRNAMDIFF, External names are different

Information: This message can appear as additional information on other error messages.

EXTRNCFLCT, "PSECT or FORWARD" conflicts with EXTERNAL visibility

Error: The storage for an external variable or routine is not allocated by the current compilation; therefore, the specification of an allocation attribute is meaningless.

FILEVALASS, FILE evaluation / assignment is not allowed

Error: You cannot attempt to evaluate a file variable or assign values to it.

FILHASSCH, FILE component may not contain nonstatic types or discriminant identifiers

Error: VAX Pascal restricts components of files to those with compile-time size.

FILOPNDREQ, FILE operand required

Error: The EOF, EOLN, and UFB functions require parameters of file types.

FILVARFIL, FILE_VARIABLE parameter must be of a FILE type

Error: The file variable parameter to the OPEN and CLOSE procedures must denote a file variable.

FLDIVPOS, Field "field name" is illegally positioned

Error: A POS attribute attempted to position a record field before the end of the previous field in the declaration.

FLDNOTKNOWN, Unknown record field

Error.

FLDONLYTXT, Field width allowed only when writing to a TEXT file
FLDWDTHINT, Field-width expression must be of type INTEGER

Error.

FORACTORD, FOR loop control variable must be of an ordinal type
FORACTVAR, FOR loop control must be a true variable

Error: The control variable of a FOR statement must be a simple variable of an ordinal type and must be declared in a VAR section. For example, it cannot be a field in a record that was specified by a WITH statement, or a function identifier.

FORCTLVAR, "variable name" is a FOR control variable

Warning: The control variable of a FOR statement cannot be assigned a value; used as a parameter to the ADDRESS function; passed as a writable VAR, %REF, %DESCR, or %STDESCR parameter; used as the control variable of a nested FOR statement; or written into by a READ, READLN, or READV procedure.

FORINEXPR, Expression is incompatible with FOR loop control variable

Error: The type of the initial or final value specified in a FOR statement is variable.

FRMLPRMDESC, Formal parameters use different descriptor formats
FRMLPRMINCMP, Formal routine parameters are not compatible
FRMLPRMNAM, Formal parameters have different names
FRMLPRMSIZ, Formal parameters have different size attributes
FRMLPRMTYP, Formal parameters have different types

Information: These messages can appear as additional information on other error messages.

FRSTPRMSTR, READV requires first parameter to be a string expression

Error: You must specify at least two parameters for the READV procedure—a character-string expression and a variable into which new values will be read.

FRSTPRMVARY, WRITEV requires first parameter to be a variable of type VARYING

Error.

FUNCTRESTYP, Routine must be declared as FUNCTION to specify a result type

Error: You cannot specify a result type on a PROCEDURE declaration.

FUNRESTYP, Function result types are different

Information: This message can appear as additional information on other error messages.

FWDREPATRLST, Declared FORWARD; repetition of attribute list not allowed
FWDREPPRMLST, Declared FORWARD; repetition of formal parameter list not allowed
FWDREPRESTYP, Declared FORWARD; repetition of result type not allowed

Error: If the heading of a routine has the FORWARD directive, the declaration of the routine body cannot repeat the formal parameter list, the result type (applies only if the routine is a function), or any attribute lists that appeared in the heading.

FWDWASFUNC, FORWARD declaration was FUNCTION
FWDWASPROC, FORWARD declaration was PROCEDURE

Error.

GOTONOTALL, GOTO not allowed to jump into a structured statement

Warning.

GTR32BITS, "routine name" cannot accept parameters larger than 32 bits

Error: DEC and UDEC cannot translate objects larger than 32 bits into their textual equivalent.

HIDATOUTER, HIDDEN legal only on definitions and declarations at outermost level

Error: When an environment file is being generated, it is possible to prevent information concerning a declaration from being included in the environment file by using the HIDDEN attribute. However, because an environment file consists only of declarations and definitions at the outermost level of a compilation unit, the HIDDEN attribute is legal only on these definitions and declarations.

IDENTGTR31, Identifier longer than 31 characters exceeds capacity of compiler

Warning.

IDNOTLAB, Identifier "symbol name" not declared as a label

Error.

IDXNOTCOMPAT, Index type is not compatible with declaration

Error: The type of an index expression is not assignment compatible with the index type specified in the array's type definition.

IDXREQDKEY, Creating INDEXED organization requires dense keys

Warning: When you specify ORGANIZATION:=INDEXED when opening a file with HISTORY := NEW or UNKNOWN, the file's alternate keys must be dense; that is, you may not omit any key numbers in the range from 0 through the highest key number specified for the file's component type.

IDXREQKEY0, Creating INDEXED organization requires FILE OF RECORD with at least KEY(0)

Warning: When you specify ORGANIZATION:=INDEXED when opening a file with HISTORY := NEW or UNKNOWN, the file's component type must be a record for which a primary key, designated by the [KEY(0)] attribute, is defined.

IMMEDBNDROU, Immediate passing mechanism may not be used on bound routine "routine name"

Warning: You cannot prefix a formal or an actual routine parameter with the immediate passing mechanism unless the routine was declared with the UNBOUND attribute.

IMMEDUNBND, Routines passed by immediate passing mechanism must be UNBOUND

Warning: A formal routine parameter that has the immediate passing mechanism must also have the UNBOUND attribute.

IMMGTR32, Immediate passing mechanism not allowed on values larger than 32 bits

Error: See Chapter 2 for more information on the types that are allocated more than 32 bits.

IMMHAVSIZ, Type passed by immediate passing mechanism must have compile-time known size

Error: You cannot specify an immediate passing mechanism for a conformant parameter or a formal parameter of type VARYING OF CHAR.

INCMPBASE, Incompatible with SET base type

Error: If no type identifier denotes the base type of a set constructor, the first element of the constructor determines the base type. The type of all subsequent elements specified in the constructor must be compatible with the type of the first.

INCMPFLDS, Record fields are not the same type

Error.

INCMPOPND, Incompatible operand(s)

Error: The types of one or more operands in an expression are not compatible with the operation being performed.

INCMPPARM, Incompatible "routine" parameter

Error: An actual routine parameter is incompatible with the corresponding formal parameter.

INCMPTAGTYP, Incompatible variant tag types

Error: This message can appear as additional information on other error messages.

INCTOODEEP, %INCLUDE directives nested too deeply, ignored

Error: A program cannot include more than five levels of files with the %INCLUDE directive. The compiler ignores %INCLUDE files beyond the fifth level.

INDNOTORD, Index type must be an ordinal type

Error: The index type of an array must be an ordinal type.

INITNOEXT, INITIALIZE routine may not be EXTERNAL
INITNOFRML, INITIALIZE routine must have no formal parameter list

Error.

INITSYNVAR, Illegal initialization syntax—Use VALUE

Error.

INPNOTDECL, INPUT not declared in heading

Error: A call to EOF, EOLN, READLN, or WRITELN did not specify a file variable, and the default INPUT or OUTPUT was not listed in the program heading.

INSTNEWLSE, Please install a new version of LSE

Error: The version of VAX LSE on your system is too old for the compiler. Contact your system manager.

INVCASERNG, Invalid range in case label list

Error.

INVEVAL, Array or Record evaluation not allowed

Error.

IVATTR, Unrecognized attribute

Error.

IVAUTOMOD, AUTOMATIC variable is illegal at the outermost level of a MODULE

Error: You cannot specify the AUTOMATIC attribute for a variable declared at module level.

IVCHKOPT, Unrecognized CHECK option

Warning.

IVCOMBFLOAT, Illegal combination of D_floating and G_floating

Error: You cannot combine D_floating and G_floating numbers in a binary operation.

IVDIRECTIVE, Unrecognized directive

Error: The directive following a procedure or function heading is not one of those recognized by the VAX Pascal compiler.

IVENVIRON, Environment "environment name" has illegal format, source analysis terminated

Fatal: The environment file inherited by the program has an illegal format; compilation is immediately aborted. However, a listing will still be produced if one was being generated.

IVFUNC, Invalid use of function "function name"
IVFUNCALL, Invalid use of function call
IVFUNCID, Invalid use of function identifier

Error: These messages result from illegal attempts to assign values or otherwise refer to the components of the function result (if its type is structured), use the type cast operator on a function identifier or its result, or deallocate the storage reserved for the function result (if its type is a pointer).

IVKEYOPT, Unrecognized KEY option

Error.

IVKEYVAL, FINDK KEY_VALUE cannot be an array (other than PACKED ARRAY [1..n] OF CHAR)

Error.

IVKEYWORD, Missing or unrecognized keyword

Error: The compiler failed to find an identifier where it expected one in a call to the OPEN or CLOSE procedure, or it found an identifier that was not legal in this position in the parameter list.

IVMATCHTYP, Invalid MATCH_TYPE parameter to FINDK

Error.

IVOPTMOPT, Unrecognized OPTIMIZE option

Warning.

IVQUALFILE, Illegal qualifier "qualifier name" on file specification

Warning: Only the /LIST and /NOLIST qualifiers are allowed on the file specification of a %INCLUDE directive.

IVQUOCHAR, Illegal nonprinting character (ASCII "nnn") within quotes

Warning: The only nonprinting characters allowed in a quoted string are the space and tab; the use of other nonprinting characters in a string causes this warning. To include nonprinting characters in a string, you should use the extended string syntax described in the *VAX Pascal Reference Manual.*

IVRADIX, Invalid radix was specified in the extended number

Error.

IVRADIXDGIT, Illegal digit in binary, octal, or hexadecimal constant

Error.

IVREDECL, Illegal redeclaration gives "symbol name" multiple meanings in "scope name"
IVREDECLREC, Illegal redeclaration gives "symbol name" multiple meanings in this record
IVREDEF, Illegal redefinition gives "symbol name" multiple meanings in "scope name"

Warning: When an identifier is used in any given block, it must have the same meaning wherever it appears in the block.

IVUSEALIGN, Invalid use of alignment attribute
IVUSEALLOC, Invalid use of allocation attribute

Error.

IVUSEATTR, Invalid use of "attribute name" attribute

Error: The use of an attribute on a variable, parameter, or routine conflicts with the requirements of the object's type.

IVUSEATTRLST, Invalid use of an attribute list

Error.

IVUSEBNDID, Illegal use of bound identifier "identifier name"

Error: An identifier that represents one bound of a conformant schema was used where a variable was expected, such as in an assignment statement or in a formal VAR parameter section. The restrictions on the use of a bound identifier are identical to those on a constant identifier.

IVUSEDES, Invalid use of descriptor class attribute

Error: The use of an attribute on a variable, parameter, or routine conflicts with the requirements of the object's type.

IVUSEFNID, Illegal use of function identifier "identifier name"

Error: Two examples of illegal uses are the assignment of values to the components of the function result (if its type is structured) and the passing of the function identifier as a VAR parameter.

IVUSESIZ, Invalid use of size attribute

Error: The use of an attribute on a variable, parameter, or routine conflicts with the requirements of the object's type.

IVUSEVISIB, invalid use of visibility attribute

Error: The use of a visibility attribute conflicts with the requirements of the object's type.

KEYINTRNG, KEY number must be an INTEGER value in range 0..254

Error: The key number specified by a KEY attribute must fall in the integer subrange 0..254.

KEYNOTALIGN, KEY "key number" field "field name" at bit position "bit position" is unaligned
KEYORDSTR, KEY allowed only on ordinal and fixed-length string fields
KEYPCKREC, KEY field in PACKED RECORD must have an alignment attribute
KEYREDECL, Key number "key number" is multiply defined
KEYSIZ1_2_4, Size of an ordinal key must be 1, 2 or 4 bytes
KEYSIZ2_4, Size of a signed integer key must be 2 or 4 bytes
KEYSIZSTR, Size of a string key cannot exceed 255 bytes
KEYUNALIGN, KEY field cannot be UNALIGNED

Error.

LABDECIMAL, Label number must be expressed in decimal radix

Error.

LABINCTAG, Variant case label's type is incompatible with tag type

Error: The type of a constant specified as a case label of a variant record is not assignment compatible with the type of the tag field.

LABNOTFND, No definition of label "label name" in statement part of "block name"

Error: A label that you declared in a LABEL section does not prefix a statement in the executable section.

LABREDECL, Redefinition of label "label name" in "block name"

Error: A label cannot prefix more than one statement in the same block.

LABRNGTAG, Variant case label does not fall within range of tag type

Error: A constant specified as a case label of a variant record is not within the range defined for the type of the tag field.

LABTOOBIG, Label "label number" is greater than MAXINT

Error.

LABUNDECL, Undeclared label "label name"

Error: VAX Pascal requires that you declare all labels in a LABEL declaration section before you use them in the executable section.

LABUNSATDECL, Unsatisfied declaration of label "label name" is not local to "block name"

Error: A label that prefixes a statement in a nested block was declared in an enclosing block.

LIBESTAB, LIB$ESTABLISH is incompatible with VAX Pascal; use predeclared procedure ESTABLISH

Warning: VAX Pascal establishes its own condition handler for processing Pascal-specific run-time signals. Calling LIB$ESTABLISH directly replaces the handler supplied by the compiler with a user-written handler; the probable result is improper handling of run-time signals. You should use Pascal's predeclared ESTABLISH procedure to establish user-written condition handlers.

LISTONEND, LIST attribute allowed only on final formal parameter

Error.

LISTUSEARG, Formal parameter has LIST attribute, use predeclared function ARGUMENT

Error: A formal parameter with the LIST attribute cannot be directly referenced. You should use the predeclared function ARGUMENT to reference the actual parameters corresponding to the formal parameter.

LNETOOLNG, Line too long, is truncated to 255 characters

Error: A source line cannot exceed 255 characters. If it does, the compiler disregards the remainder of the line.

LOWGTRHIGH, Low-bound exceeds high-bound

Error: The definition of the flagged subrange type is illegal because the value specified for the lower limit exceeds that for the upper limit.

MAXLENINT, Max-length must be a value of type INTEGER

Error: The maximum length specified for type VARYING OF CHAR must be an integer in the range 1..65535; that is, the type definition must denote a legal character string.

MAXLENRNG, Max-length must be in range 1..65535

Error: The maximum length specified for type VARYING OF CHAR must be an integer in the range 1..65535; that is, the type definition must denote a legal character string.

MAXNUMENV, Maximum number of environments exceeded

Fatal: More than 512 environment files were used in the compilation.

MECHEXTERN, Foreign mechanism specifier allowed only on external routines

Error.

MISSINGEND, No matching END, expected near line "line number"

Information: The compiler expected an END statement at a location where none was found. Compilation proceeds as though the END statement were correctly located.

MODINIT26, Module name limited to 26 characters when initialization required

Error: When a module contains schema types, discriminated schema types, variables of discriminated schema types, or a TO BEGIN DO statement clause, the module name is limited to 26 characters.

MODOFNEGNUM, MOD of a negative modulus has no mathematical definition

Error: In the MOD operation A MOD B, the operand B must have a positive integer value. This message is issued only when the MOD operation occurs in a compile-time constant expression.

MSTBEARRAY, Type must be ARRAY

Error.

MSTBEARRVRY, Type must be ARRAY or VARYING

Error: You cannot use the syntax [index] to refer to an object that is not of type ARRAY or VARYING OF CHAR.

MSTBEBOOL, Control expression must be of type BOOLEAN

Error: The IF, REPEAT, and WHILE statements require a Boolean control expression.

MSTBEDEREF, Must be dereferenced

Information.

MSTBEDISCR, Schema type must be discriminated

Error: An undiscriminated schema type is not allowed everywhere that a regular type name is allowed.

MSTBEORDSETARR, Type must be ordinal, SET, or ARRAY

Error.

MSTBEREC, Type must be RECORD

Error.

MSTBERECVRY, Type must be RECORD or VARYING

Error: You cannot use the syntax "Variable.Identifier" to refer to an object that is not of type RECORD or VARYING OF CHAR.

MSTBESTAT, Cannot initialize non-STATIC variables

Error: You cannot initialize variables declared without the STATIC attribute in nested blocks, nor can you initialize program-level variables whose attributes give them some allocation other than static.

MSTBETEXT, "I/O routine" requires FILE_VARIABLE of type TEXT

Error: The READLN and WRITELN procedures operate only on text files.

MULTDECL, "symbol name" has multiple conflicting declarations, reason(s):

Error.

NCATOA, Cannot reformat content of actual's CLASS_NCA descriptor as CLASS_A

Error: This message can appear as additional information on other error messages.

NEWQUADAGN, "type name"'s base type is ALIGNED("nnn"); NEW handles at most ALIGNED(3)

Error: You cannot call the NEW procedure to allocate pointer variables whose base types specify alignment greater than a quadword. To allocate such variables, you must use external routines.

NOACTCOM, No actuals are compatible with schema formal parameter

Information: Undiscriminated schema formal parameters denoting subranges or sets cannot be used as value parameters. In these cases, no actual parameter can ever be compatible with the formal parameter.

NOASSTOFNC, Block does not contain an assignment to function result "function name"

Warning: The block of a function must include a statement that assigns a return value to the function identifier.

NOCONVAL, A constant value was not specified for field "field name"

Error.

NODECLVAR, "symbol name" is not declared in a VAR section of "block name"

Error: You cannot initialize a variable using the VALUE section if the variable was not declared in the same block in which the VALUE section appears.

NODSCREC, No descriptor class for RECORD type

Error: The VAX Procedure Calling Standard does not define a descriptor format for records; therefore, you cannot specify %DESCR for a parameter of type RECORD.

NODSCRSCH, No descriptor class for schematic types

Error.

NOFLDREC, No field "field name" in RECORD type "type name"

Error: The field specified does not exist in the specified record.

NOFRMINDECL, Declaration of "routine" parameter "routine name" supplied no formal parameter list

Information: You specified actual parameters in a call on a formal routine parameter that was declared with no formal parameters. Although such a call was legal in VAX Pascal Version 1.0, it does not follow the rules of the Pascal standard. You should edit your program to reflect this change.

NOINITEXT, Initialization not allowed on EXTERNAL variables
NOINITINH, Initialization not allowed on inherited variables

Error: You can initialize only those variables whose storage is allocated in this compilation.

NOINITVAR, Cannot initialize "symbol name"—it is not declared as a variable

Error: Variables are the only data items that can be initialized, and they can be initialized only once.

NOLISTATTR, Parameter to this predeclared function must have LIST attribute

Error: ARGUMENT and ARGUMENT_LIST_LENGTH require their first parameter to be a formal parameter with the LIST attribute.

NOREPRE, No textual representation for values of this type

Error: You cannot write a value to a text file (using WRITE or WRITELN) or to a VARYING string (using WRITEV) if there is no textual representation for the type. Similarly, you cannot read a value from a text file (using READ or READLN) or from a VARYING string (using READV) if there is no textual representation for the type. Such types are RECORD, ARRAY (other than PACKED ARRAY [1..n] OF CHAR), SET, and pointer.

NOTAFUNC, "symbol name" is not declared as a "routine."

Error: An identifier followed by a left parenthesis, a semicolon, or one of the reserved words END, UNTIL, and ELSE is interpreted as a call to a routine with no parameters. This message is issued if the identifier was not declared as a procedure or function identifier. Note that in the current version, functions can be called with the procedure call statement.

NOTASYNCH, "routine name" is not ASYNCHRONOUS

Information: This message can appear as additional information on other error messages.

NOTATYPE, "symbol name" is not a type identifier

Error: An identifier that does not represent a type was used in a context where the compiler expected a type identifier.

NOTAVAR, "symbol name" is not declared as a variable

Error: You cannot assign a value to any object other than a variable.

NOTAVARFNID, "symbol name" is not declared as a variable or a function identifier

Error: You cannot assign a value to any object other than a variable or a function identifier.

NOTAVARPARM, "symbol name" is not declared as a variable or parameter

Error.

NOTBEADDR, May not be parameter to ADDRESS
NOTBEASSIGN, May not be assigned
NOTBECALL, May not be called as a FUNCTION
NOTBECAST, May not be type cast
NOTBEDEREF, May not be dereferenced
NOTBEDES, May not be passed by untyped %DESCR
NOTBEEVAL, May not be evaluated
NOTBEFILOP, May not be used in a file operation

NOTBEFLD, May not be field selected
NOTBEFNCPRM, May not be passed as a FUNCTION parameter
NOTBEFORCTL, May not be used as FOR loop variable
NOTBEFORDES, May not be passed as a descriptor foreign parameter
NOTBEFOREF, May not be passed as a reference foreign parameter
NOTBEIADDR, May not be parameter to IADDRESS
NOTBEIDX, May not be indexed
NOTBEIMMED, May not be passed by untyped immediate passing mechanism

NOTBENEW, May not be written into be NEW
NOTBENSTCTL, May not be control variable for an inner FOR loop
NOTBEREAD, May not be written into be READ
NOTBEREF, May not be passed by untyped reference passing mechanism
NOTBERODES, May not be passed as a READONLY descriptor foreign parameter
NOTBEROFOR, May not be passed as a READONLY reference foreign parameter
NOTBEROVAR, May not be passed as a READONLY VAR parameter
NOTBETOUCH, May not be read/modified/written

NOTBEVAR, May not be passed as a VAR parameter
NOTBEWODES, May not be passed as a WRITEONLY descriptor foreign parameter
NOTBEWOFOR, May not be passed as a WRITEONLY reference foreign parameter
NOTBEWOVAR, May not be passed as a WRITEONLY VAR parameter
NOTBEWRTV, May not be parameter to WRITEV

Information: These messages can appear as additional information on other error messages.

NOTBYTOFF, Field "field name" is not aligned on a byte boundary

Error.

NOTDECLROU, "symbol name" is not declared as a "routine."
NOTINITIAL, "routine name" is not INITIALIZE

Information: These messages can appear as additional information on other error messages.

NOTINRNG, Value does not fall within range of the tag type

Error: The value specified as the case label of a variant record is not a legal value of the tag field's type. This message is also issued if a case label in a call to NEW, DISPOSE, or SIZE falls outside the range of the tag type.

NOTNEWTYP, Schema must define a new type

Error: The type-denoter of a schema definition must define a new type; for example, a subrange, an array, or a record.

NOTSAMTYP, Not the same type
NOTUNBOUND, "routine name" is not UNBOUND

Information: These messages can appear as additional information on other error messages.

NOTSCHEMA, "symbol name" is not a schema type

Error.

NOTVARNAM, Parameter to this predeclared function must be simple variable name

Error: The parameter cannot be indexed, be dereferenced, have a field selected, or be an expression. It must be the name of the entire variable.

NOTVOLATILE, "variable name" is non-VOLATILE

Warning: You should not use the ADDRESS function on a non volatile variable or component or on a formal VAR parameter.

NOUNSATDECL, No unsatisfied declaration of label "label name" in "block name"

Error.

NUMFRMLPARM, Different numbers of formal parameters

Information: This message can appear as additional information on other error messages.

NXTACTDIFF, NEXT of actual's component differs from that of other parameters in same section

Error: All actual parameters passed to a formal parameter section whose type is a conformant schema must have identical bounds and be structurally compatible. This message refers to the allocation size and alignment of the array's inner dimensions.

OLDDECLSYN, Obsolete "routine" parameter declaration syntax

Information: The declaration of a formal routine parameter uses the obsolete Version 1.0 syntax. You should edit your program to incorporate the current version syntax, which is mandated by the Pascal standard.

OPNDASSCOM, Operands are not assignment compatible
OPNOTINT, Operand(s) must be of type INTEGER or UNSIGNED

Error.

OPNDNAMCOM, Operands are not name compatible

Error.

ORDOPNDREQ, Ordinal operand(s) required

Error or Warning: This message is at warning level if you try to use INT, ORD, or UINT on a pointer expression. It is at error level if you use PRED or SUCC on an expression whose type is not ordinal.

OUTNOTDECL, OUTPUT not declared in heading

Error: A call to EOF, EOLN, READLN, or WRITELN did not specify a file variable, and the default INPUT or OUTPUT was not listed in the program heading.

OVRDIVZERO, Overflow or division by zero in compile-time expression

Error.

PACKSTRUCT, "component name" of a PACKED structured type

Error or Warning: You cannot use the data items listed in a call to the ADDRESS function, nor can you pass them as writable VAR, %REF, %DESCR, or %STDESCR parameters. This message is at warning level if the variable or component has the UNALIGNED attribute, and at error level if the variable or component is actually unaligned.

PARMACTDEF, Formal parameter "parameter name" has neither actual nor default

Error: If a formal parameter is not declared with a default, you must pass an actual parameter to it when calling its routine.

PARMCLAMAT, Parameter section classes do not match

Information: This message can appear as additional information on other error messages.

PARMLIMIT, VAX architectural limit of 255 parameters exceeded

Error: You cannot declare a procedure with more than 255 formal parameters. A function whose result type requires that the result be stored in more than 64 bits or whose result type is a character string cannot have more than 254 formal parameters. In a call to a routine declared with the LIST attribute, you also cannot pass more than 255 (or 254) actual parameters.

PARMSECTMAT, Division into parameter sections does not match

Information: This message can appear as additional information on other error messages.

PARSEFAIL, error parsing command line; use PASCAL command

Fatal: The VAX Pascal compiler was invoked without using the PASCAL DCL command.

PARSEFAIL, error parsing command line; using an invalid CLD table

Fatal: The VAX Pascal compiler was invoked with an incorrect or obsolete command line definition in SYS$LIBRARY:DCLTABLES. Contact your system manager to reinstall SYS$LIBRARY: DCLTABLES.

PASPREILL, Passing predeclared "routine name" is illegal

Error: You cannot use the IADDRESS function on a predeclared routine for which there is no corresponding routine in the VMS Run-Time Library (such as the interlocked functions). In addition, you cannot pass a predeclared routine as a parameter if there is no way to write the predeclared routine's formal parameter list in VAX Pascal. Examples of the latter case are the PRED and SUCC functions and many of the I/O routines.

PASSEXTERN, Passing mechanism allowed only on external routines

Error.

PCKARRBOO, PACKED ARRAY OF BOOLEAN parameter expected

Error.

PCKUNPCKCON, Packed/unpacked conflict

Information: This message can appear as additional information on other error messages.

PLACEBEFEOLN, Placeholder not terminated before end of line

Error.

PLACEIVCHAR, Illegal nonprinting character (ASCII "decimal representation of character") within placeholder

Warning.

PLACENODOT, Repetition of pseudocode placeholders not allowed

Error.

PLACESEEN, Placeholder encountered

Error.

PLACEUNMAT, Unmatched placeholder delimiter

Error.

POSAFTNONPOS, Positional parameter cannot follow a nonpositional parameter

Error.

POSALIGNCON, Position / alignment conflict

Error: The bit position specified by the POS attribute does not have the number of low-order bits implied by the specified alignment attribute.

POSINT, POS expression must be a positive INTEGER value

Error.

PREREQPRMLST, Passing predeclared "routine name" requires formal to include parameter list

Error: To pass one of the predeclared routines EXPO, ROUND, TRUNC, UNDEFINED, UTRUNC, UROUND, DBLE, SNGL, QUAD, INT, ORD, and UINT as an actual parameter to a routine, you must specify a formal parameter list in the corresponding formal routine parameter.

PRMKWNSIZ, Parameter must have a size known at compile-time

Error: The BIN, HEX, OCT, DEC, and UDEC functions cannot be used on conformant parameters. The SIZE and NEXT functions cannot be used on conformant parameters in compile-time constant expressions.

PROCESSFILE, Compiling file "file name"

Information.

PROCESSRTN, Generating code for routine "routine name"

Information.

PROGSCHENV, PROGRAM with schema may not create environment

Error: A program that declares a schema type cannot have the [ENVIRONMENT] attribute. Schema declarations should be placed in a separate module and inherited by the program.

PROPRMEXT, Declaration of "program parameter name" is EXTERNAL—program parameter files must be locally allocated

PROPRMFIL, A program parameter must be a variable of type FILE

PROPRMINH, Declaration of "program parameter name" is inherited—program parameter files must be locally allocated

PROPRMLEV, Program parameter "program parameter name" is not declared as a variable at the outermost level

Error: Any external file variable (other than INPUT and OUTPUT) that is listed in the program heading must also be declared as a file variable in a VAR section in the program block.

PSECTMAXINT, Allocation of "symbol name" causes PSECT "PSECT name" to exceed MAXINT bits

Error: The VAX implementation restricts the size of a program section to 2,147,483,647 bits.

PTRCMPEQL, Pointer values may only be compared for equality

Error: The equality (=) and inequality (<>) operators are the only operators allowed for values of a pointer type; all other operators are illegal.

PTREXPRCOM, Pointer expressions are not compatible

Error: The base types of two pointer expressions being compared for equality (=) or inequality (<>) are not structurally compatible.

QUOBEFEOL, Quoted string not terminated before end of line

Error.

QUOSTRING, Quoted string expected

Error: The compiler expects the %DICTIONARY and %INCLUDE directives, and the radix notations for binary (%B), hexadecimal (%X), and octal constants (%O), to be followed by a quoted string of characters.

RADIXTEXT, Radix input requires FILE_VARIABLE of type TEXT

Error: The input radix specifiers (BIN, OCT, and HEX) operate only on text files.

READONLY, "variable name" is READONLY

Warning: You cannot use a read-only variable in any context that would store a new value in the variable. For example, a read-only variable cannot be used in a file operation.

REALCNSTRNG, Real constant out of range

Error: See the *VAX Pascal Reference Supplement for VMS Systems* for details on the range of real numbers.

REALOPNDREQ, Real (SINGLE, DOUBLE or QUADRUPLE) operand(s) required

Error.

RECHASFILE, Record contains one or more FILE components, POS is illegal

Error.

RECHASTMSTMP, Record contains one or more TIMESTAMP components, POS is illegal

Error.

RECLENINT, RECORD_LENGTH expression must be of type INTEGER

Error: The value of the record length parameter to the OPEN procedure must be an integer.

RECLENMNGLS, RECORD_LENGTH parameter is meaningless given file's type

Warning: The record length parameter is usually relevant only for files of type TEXT and VARYING OF CHAR.

RECMATCHTYP, MATCH_TYPE identifier "NXT or NXTEQL" is recommended instead of "GTR or GEQ"

Information.

REDECL, A declaration of "symbol name" already exists in "block name"

Error: You cannot redeclare an identifier or a label in the same block in which it was declared. Inheriting an environment is equivalent to including all of its declarations at program or module level.

REDECLATTR, "attribute name" already specified

Error: Only one member of a particular attribute class can appear in the same attribute list.

REDECLFLD, Record already contains a field "field name"

Error: The names of the fields in a record must be unique; they cannot be duplicated between variants.

REINITVAR, "variable name" has already been initialized

Error: Variables are the only data items that can be initialized, and they can be initialized only once.

REPCASLAB, Value has already appeared as a label in this CASE statement

Error: You cannot specify the same value more than once as a case label in a CASE statement.

REPFACZERO, Repetition factor cannot be the function ZERO
REQCLAORNCA, Arrays and conformants of this parameter type require either CLASS_A or CLASS_NCA
REQCLS, Scalars and strings of this parameter type require CLASS_S

Error.

REQNOCH, Primary key requires NOCHANGES option

Error.

REQPKDARR, The combination of CLASS_S and %STDESCR requires a PACKED ARRAY OF CHAR structure

Error.

REQREADVAR, READ or READV requires at least one variable to read into

Error: The READ and READV procedures require that you specify at least one variable to be read from a file.

REQWRITELEM, WRITE requires at least one write-list-element

Error: The WRITE procedure requires that you specify at least one item to be written to a file.

RESPTRTYP, Result must be a pointer type

Information.

REVRNTLAB, Value has already appeared as a label in this variant part

Error: You cannot specify the same value more than once as a case label in a variant part of a record.

RTNSTDESCR, Routines cannot be passed using %STDESCR

Error.

SCHCONST, Nonstatic constants are not allowed

Error: Constants cannot be made for nonstatic types since that would yield constants without compile-time size and value.

SCHFLDALN, Field in nonstatic type may not have greater than byte alignment

Error.

SCHOVERLAID, Use of schema types conflicts with OVERLAID attribute

Error: The OVERLAID attribute cannot be used on programs or modules that discriminate schema at the outermost level.

SENDSPR, Internal Compiler Error

Fatal: An error has occurred in the execution of the VAX Pascal compiler. Along with this message, you will receive information that helps you find the location in the source program and the name of the compilation phase at which the error occurred. You may be able to rewrite the section of your program that caused the error and thus successfully compile the program. However, even if you are able to remedy the problem, please submit a Software Performance Report (SPR) to Digital and provide a machine-readable copy of the program.

SEQ11FORT, PDP–11 specific directive SEQ11 treated as equivalent to FORTRAN directive

Information.

SETBASCOM, SET base types are not compatible

Error: The base type of two sets used in a set operation are not compatible.

SETELEORD, SET element expression must be of an ordinal type

Error: The expressions used to denote the elements of a set constructor or the bounds of a set type definition must have an ordinal type.

SETNOTRNG, SET element is not in range 0..255

Error: In a set whose base type is a subrange of integers or unsigned integers, all set elements in the set's type definition or in a constructor for the set must be in the range 0..255.

SIZACTDIFF, SIZE of actual differs from that of other parameters in same section

Error: All actual parameters passed to a formal parameter section whose type is a conformant schema must have identical bounds and be structurally compatible. This message refers to the allocation size of the array's outermost dimension.

SIZARRNCA, Explicit size on ARRAY dimension makes CLASS_NCA mandatory

Error.

SIZATRTYPCON, Size attribute / type conflict

Error: For an ordinal type, the size specified must be at least as large as the packed size but no larger than 32 bits. Pointer types and type SINGLE must be allocated exactly 32 bits, type DOUBLE exactly 64 bits, and type QUADRUPLE exactly 128 bits. For types ARRAY, RECORD, SET, and VARYING OF CHAR, the size specified must be at least as large as their packed sizes. For the details of allocation sizes in VAX Pascal, see Chapter 2.

SIZCASTYP, Variable's size conflicts with cast's target type

Error: In a type cast operation, the size of the variable and the size of the type to which it is cast must be identical.

SIZEDIFF, Sizes are different

Information: This message can appear as additional information on other error messages.

SIZEINT, Size expression must be a positive INTEGER value

Error.

SIZGTRMAX, Size exceeds MAXINT bits

Error: The size of a record or an array type or the size specified by a size attribute exceeds 2,147,483,647 bits. The VAX implementation imposes this size restriction.

SIZMULTBYT, Size of component of array passed by descriptor is not a multiple of bytes

Error: When an array or a conformant parameter is passed using the %DESCR mechanism specifier, the descriptor built by the compiler must follow the VAX Procedure Calling Standard. Such a descriptor can describe only an array whose components fall on byte boundaries.

SPEOVRDECL, Foreign mechanism specifier required to override parameter declaration

Error: When you specify a default value for a formal VAR or routine parameter, you must also use a mechanism specifier to override the characteristics of the parameter section.

SPURIOUS, "error message" at "line number"—"column number"

Information: The compiler did not correctly note the location of this error in your program and later could not position and print the correct error message. You may be able to correct the section of your program that caused the error and thus avoid this error. Please submit a Software Performance Report (SPR) and provide a machine-readable copy of the program if you receive this error.

SRCERRORS, Source errors inhibit continued compilation—correct and recompile

Fatal: A serious error previously detected in the source program has corrupted the compiler's symbol tables and inhibits further compilation. Correct the serious error and recompile the program.

SRCTXTIGNRD, Source text following end of compilation unit ignored

Warning: The compiler ignores any text following the END statement that terminates a compilation unit. This error probably resulted from an unmatched END statement in your program.

STDACTINCMP, Nonstandard: actual is not name compatible with other parameters in same section

Information: According to the Pascal standard, all actual parameters passed to a parameter section must have the same type identifier or the same type definition. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDATTRLST, Nonstandard: attribute list
STDBIGLABEL, Nonstandard: label number greater than 9999
STDBLANKPAD, Nonstandard: blank-padding used during string operation
STDBNDRMUSE, Nonstandard: usage of formal parameter for routine "routine name"
STDCALLFUNC, Nonstandard: function "function name" called as a procedure
STDCASLBLRNG, Nonstandard: label range in case selector

STDCAST, Nonstandard: type cast operator
STDCMPCOMPAT, Nonstandard: cannot "PACK or UNPACK", array component types are incompatible
STDCMPDIR, Nonstandard: compiler directive
STDCOMFUNACC, Nonstandard: component function access
STDCNFARR, Nonstandard: conformant array syntax

Information: These messages refer to VAX extensions to Pascal and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDCNSTR, Nonstandard: array or record constructor

Information: A VAX Pascal style constructor was used. You should convert this constructor to the new constructor syntax provided in VAX Pascal Version 4.0 to be compatible with the Extended Pascal standard.

STDCONCAT, Nonstandard: concatenation operator

Information: This message refers to VAX extensions to Pascal and is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDCONST, Nonstandard: "type name" constant

Information: Binary, hexadecimal, and octal constants and constants of type DOUBLE, QUADRUPLE, and UNSIGNED are VAX extensions to Pascal. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDCONSTACC, Nonstandard: structured constant access

Information: This message is issued if you have specified a /STANDARD option other than /STANDARD=EXTENDED with the PASCAL command.

STDCTLDECL, Nonstandard: control variable "variable name" not declared in VAR section of "block name"

Information: The Pascal standard requires that the control variable of a FOR statement be declared in the same block in which the FOR statement appears.

STDDECLSEC, Nonstandard: declaration sections either out of order or duplicated in "block name"

Information: In the Pascal standard, the declaration sections must appear in the order LABEL, CONST, TYPE, VAR, PROCEDURE, and FUNCTION. The ability to specify the sections in any order is a VAX extension. This message occurs only if you have specified the /STANDARD qualifier with the PASCAL command.

STDDEFPARM, Nonstandard: default parameter declaration

Information: This message refers to VAX extensions to Pascal and is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDDIRECT, Nonstandard: "directive name" directive

Information: The EXTERN, EXTERNAL, FORTRAN, and SEQ11 directives are VAX extensions to Pascal. (FORWARD is the only directive specified by the Pascal standard.) This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDDISCREF, Nonstandard: schema discriminant reference
STDDISCSCHEMA, Nonstandard: discriminated schema

Information: These messages are issued if you have specified a /STANDARD option other than /STANDARD=EXTENDED with the PASCAL command.

STDEMPCASLST, Nonstandard: empty case-list element

Information: This message is issued if you do not specify any case labels and executable statements between two semicolons or between OF and a semicolon in the CASE statement. You must also have specified the /STANDARD qualifier with the PASCAL command.

STDEMPPARM, Nonstandard: empty actual parameter position

Information: This message refers to VAX extensions to Pascal and is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDEMPREC, Nonstandard: empty record section

Information: The Pascal standard does not allow record type definitions of the form RECORD END. This message appears only if you have specified the /STANDARD qualifier with the PASCAL command.

STDEMPSTR, Nonstandard: empty string

Information: This message refers to VAX extensions to Pascal and is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDEMPVRNT, Nonstandard: empty variant

Information: This message occurs if you do not specify a variant between two semicolons or between OF and a semicolon. You must also have specified the /STANDARD qualifier with the PASCAL command.

STDERRPARM, Nonstandard: error-recovery parameter
STDEXPON, Nonstandard: exponentiation operator
STDEXTSTR, Nonstandard: extended string syntax

Information: These messages refer to VAX extensions to Pascal and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDFLDHIDPTR, Nonstandard: record field identifier "field identifier name" hides type identifier "field identifier name"

Information.

STDFORIN, Nonstandard: SET-iteration in FOR statement

Information: This message is issued if you have specified a /STANDARD option other than /STANDARD=EXTENDED with the PASCAL command.

STDFORMECH, Nonstandard: foreign mechanism specifier

Information: This message refers to a VAX extension to Pascal and is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDFORWARD, Nonstandard: PROCEDURE/FUNCTION block "routine name" and it's FORWARD heading are not in the same section

Information: The Extended Pascal standard requires that FORWARD declared routines must specify their corresponding blocks without intervening LABEL, CONST, TYPE, or VAR sections. This message is issued only if you have specified the /STANDARD=EXTENDED qualifier with the PASCAL command.

STDFUNIDEVAR, Nonstandard: function identified variable

Information: This message is issued if you have specified a /STANDARD option other than /STANDARD=EXTENDED with the PASCAL command.

STDFUNCTRES, Nonstandard: FUNCTION returning a value of a "type name" type

Information: The ability of functions to have structured result types is a VAX extension to Pascal. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDINCLUDE, Nonstandard: %INCLUDE directive
STDINITVAR, Nonstandard: initialization syntax in VAR section

Information: These messages refer to VAX extensions to Pascal and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDKEYWRD, Nonstandard: "keyword name"

Information: This message is issued if you have specified a /STANDARD option other than /STANDARD=EXTENDED with the PASCAL command.

STDMATCHVRNT, Nonstandard: no matching variant label

Information: This message is issued if you call the NEW or DISPOSE procedure, and one of the case labels specified in the call does not correspond to a case label in the record variable. You must also have specified the /STANDARD qualifier with the PASCAL command.

STDMODCTL, Nonstandard: potential uplevel modification of "variable name" prohibits use as control variable

Information: You cannot use as the control variable of a FOR statement any variable that might be modified in a nested block. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDMODULE, Nonstandard: MODULE declaration

Information: The item listed in this message is a VAX extension to Pascal. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDNILCON, Nonstandard: use of reserved word NIL as a constant

Information: Only simple constants and quoted strings are allowed by the Pascal standard to appear as constants. Simple constants are integers, character strings, real constants, symbolic constants, and constants of BOOLEAN and enumerated types. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDNOFRML, Nonstandard: FUNCTION or PROCEDURE parameter declaration lacks formal parameter list

Information: This message is issued if you try to pass actual parameters to a formal routine parameter for which you declared no formal parameter list. You must also have specified the /STANDARD qualifier with the PASCAL command.

STDNONPOS, Nonstandard: nonpositional parameter syntax
STDOTHER, Nonstandard: OTHERWISE clause
STDPASSPRE, Nonstandard: passing predeclared "routine name"

Information: These messages refer to VAX extensions to Pascal and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDPCKSET, Nonstandard: combination of packed and unpacked sets

Information: The Pascal standard does not allow packed and unpacked sets to be combined in set operations. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDPRECONST, Nonstandard: predeclared constant "constant name"

Information: The constants MAXCHAR, MAXUNSIGNED, MAXREAL, MINREAL, EPSREAL, MAXDOUBLE, MINDOUBLE, EPSDOUBLE, MAXQUADRUPLE, MINQUADRUPLE, and EPSQUADRUPLE are VAX extensions to Pascal. MAXCHAR, MAXREAL, MINREAL, and EPSREAL are contained in Extended Pascal. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDPREDECL, Nonstandard: predeclared "routine"

Information: Many predeclared procedures and functions are VAX extensions to Pascal. The use of these routines causes this message to be issued if you have specified the /STANDARD qualifier with the PASCAL command.

STDPRESCH, Nonstandard: predefined schema "type name"

Information: This message is issued if you have specified a /STANDARD option other than /STANDARD=EXTENDED with the PASCAL command.

STDPRETYP, Nonstandard: predefined type "type name"

Information: The types SINGLE, DOUBLE, QUADRUPLE, UNSIGNED, and VARYING OF CHAR are VAX extensions to Pascal. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDQUOSTR, Nonstandard: quotes enclosing radix constant

Information: This message is issued if you have specified the /STANDARD=EXTENDED option with the PASCAL command.

STDRADFORMAT, Nonstandard: use format "radix"#nnn for radix constant

Information: This message refers to the use of a VAX extension to Pascal. This message is issued only if you have specified the /STANDARD=EXTENDED with the PASCAL command.

STDRADIX, Nonstandard: radix constant

Information: This message refers to the use of a VAX extension to Pascal. This message is issued only if you have specified a /STANDARD option other than /STANDARD=EXTENDED with the PASCAL command.

STDRDBIN, Nonstandard: binary input from a TEXT file
STDRDENUM, Nonstandard: enumerated type input from a TEXT file
STDRDHEX, Nonstandard: hexadecimal input from a TEXT file
STDRDOCT, Nonstandard: octal input from a TEXT file
STDRDSTR, Nonstandard: string input from a TEXT file

Information: The Pascal standard allows only INTEGER, CHAR, and REAL values to be read from a text file. The ability to read values of other types is a VAX extension to Pascal. These messages are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDREDECLNIL, Nonstandard: redeclaration of reserved word NIL

Information: The Pascal standard considers NIL a reserved word, while VAX Pascal considers it to be a predeclared identifier. Thus, if you have specified the /STANDARD qualifier with the PASCAL command, this message will be issued if you attempt to redefine NIL.

STDREM, Nonstandard: REM operator

Information: The item listed in this message is a VAX extension to Pascal. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDSCHEMA, Nonstandard: schema type definition

Information: This message is issued if you have specified a /STANDARD option other than /STANDARD=EXTENDED with the PASCAL command.

STDSCHEMAUSE, Nonstandard: use of schema type

Information: This message is issued if you have specified a /STANDARD option other than /STANDARD=EXTENDED with the PASCAL command.

STDSIMCON, Nonstandard: only simple constant (optional sign) or quoted string

Information: Only simple constants and quoted strings are allowed by the Pascal standard to appear as constants. Simple constants are integers, character strings, real constants, symbolic constants, constants of type BOOLEAN, and enumerated types. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDSPECHAR, Nonstandard: "$" or "_" in identifier
STDSTRCOMPAT, Nonstandard: string compatibility

Information: These messages refer to VAX extensions to Pascal and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDSTRUCT, Nonstandard: types do not have same name

Information: Because the Pascal standard does not recognize structural compatibility, two types must have the same type identifier or type definition to be compatible. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDSYMLABEL, Nonstandard: symbolic label

Information: These messages refer to VAX extensions to Pascal and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDTAGFLD, Nonstandard: invalid use of tag field

Information: The tag field of a variant record cannot be a parameter to the ADDRESS function, nor can you pass it as a writable VAR, %REF, %DESCR, or %STDESCR formal parameter. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDTODECL, Nonstandard: TO BEGIN/END DO declaration

Information: This message is issued if you have specified a /STANDARD option other than /STANDARD=EXTENDED with the PASCAL command.

STDUNSAFE, Nonstandard: UNSAFE compatibility

Information: If you have used the UNSAFE attribute on an object that is later tested for compatibility, you will receive this message. You must also have specified the /STANDARD qualifier with the PASCAL command.

STDUSEDCNF, Nonstandard: conformant array used as a string
STDUSEDPCK, Nonstandard: PACKED ARRAY [1..1] OF CHAR used as a string

Information: These messages refer to VAX extensions to Pascal and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDVALCNFPRM, Nonstandard: conformant array may not be passed to value conformant parameter

Information.

STDVALUE, Nonstandard: VALUE initialization section
STDVAXCDD, Nonstandard: %DICTIONARY directive

Information: These messages refere to VAX extensions to Pascal and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDVRNTCNSTR, Nonstandard: variant field outside constructor variant part

Information: This message refers to the use of a VAX extension to Pascal. This message is issued only if you have specified the /STANDARD=EXTENDED with the PASCAL command.

STDVRNTPART, Nonstandard: empty variant part

Information: According to the Pascal standard, a variant part that declares no case labels and field lists between the words OF and END is illegal. This message occurs only if you have specified the /STANDARD qualifier with the PASCAL command.

STDVRNTRNG, Nonstandard: variant labels do not cover the range of the tag type

Information: According to the Pascal standard, you must specify one case label for each value in the tag type of a variant record. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDWRTBIN, Nonstandard: binary output to a TEXT file
STDWRTENUM, Nonstandard: user defined enumerated type output to a TEXT file
STDWRTHEX, Nonstandard: hexadecimal output to a TEXT file
STDWRTOCT, Nonstandard: octal output to a TEXT file

Information: The Pascal standard allows only INTEGER, BOOLEAN, CHAR, REAL, and PACKED ARRAY [1..n] OF CHAR values to be written to a text file. The ability to write values of other types is a VAX extension to Pascal. These messages are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STREQLLEN, String values must be of equal length

Error: You cannot perform string comparisons on character strings that have different lengths.

STROPNDREQ, String (CHAR, PACKED ARRAY [1..n] OF CHAR, or VARYING) operand required
STRPARMREQ, String (CHAR, PACKED ARRAY [1..n] OF CHAR, or VARYING) parameter required
STRTYPREQ, String (CHAR, PACKED ARRAY [1..n] OF CHAR, or VARYING) type required

Error: The file name parameter to the OPEN procedure and the parameter to the LENGTH function must be character strings of the types listed.

SYNASCII, Illegal ASCII character
SYNASSERP, Syntax: ":=", ";" or ")" expected
SYNASSIGN, Syntax: ":=" expected
SYNASSIN, Syntax: ":=" or IN expected
SYNASSSEMI, Syntax: ":=" or ";" expected
SYNATRCAST, Syntax: attribute list not allowed on a type cast
SYNATTTYPE, Syntax: attribute-list or type specification
SYNBEGDECL, Syntax: BEGIN or declaration expected

SYNBEGEND, Syntax: BEGIN or END expected
SYNBEGIN, Syntax: BEGIN expected
SYNCOASSERP, Syntax: ",", ":=", ";" or ")" expected
SYNCOELRB, Syntax: ",", ".." or "]" expected
SYNCOLCOMRP, Syntax: ":", "," or ")" expected
SYNCOLON, Syntax: ":" expected
SYNCOMCOL, Syntax: "," or ":" expected
SYNCOMDO, Syntax: "," or DO expected

SYNCOMEQL, Syntax: "," or "=" expected

SYNCOMMA, Syntax: "," expected
SYNCOMRB, Syntax: "," or "]" expected
SYNCOMRP, Syntax: "," or ")" expected
SYNCOMSEM, Syntax: "," or ";" expected
SYNCONTMESS, Syntax: CONTINUE or MESSAGE expected
SYNCOSERP, Syntax: ",", ";" or ")" expected
SYNDIRBLK, Syntax: directive or block expected

> **Error:** The compiler either failed to find an important lexical or syntactical element where one was expected, or it detected an error in such an element that does exist in your program.

SYNDIRMIS, Syntax: directive missing, EXTERNAL assumed

> **Error:** In the absence of a directive where one is expected, the compiler assumes that EXTERNAL is the intended directive and proceeds with compilation based on that assumption.

SYNDO, Syntax: DO expected
SYNELIPSIS, Syntax: ".." expected
SYNELSESTMT, Syntax: ELSE or start of new statement expected
SYNEND, Syntax: END expected
SYNEQL, Syntax: "=" expected
SYNEQLLP, Syntax: "=" or "(" expected
SYNERRCTE, Error in compile-time expression
SYNEXPR, Syntax: expression expected
SYNEXSEOTEN, Syntax: expression, ";", OTHERWISE or END expected

SYNFUNPRO, Syntax: FUNCTION or PROCEDURE expected
SYNHEADTYP, Syntax: routine heading or type identifier expected
SYNIDCAEND, Syntax: identifier, CASE or END expected
SYNIDCARP, Syntax: identifier, CASE or ")" expected
SYNIDCASE, Syntax: identifier or CASE expected
SYNIDENT, Syntax: identifier expected
SYNILLEXPR, Syntax: ill-formed expression
SYNINT, Syntax: integer expected

SYNINVSEP, Syntax: invalid token separator
SYNIVATRLST, Syntax: illegal attribute list
SYNIVPARM, Syntax: illegal actual parameter
SYNIVPRMLST, Syntax: illegal actual parameter list
SYNIVSYM, Syntax: illegal symbol
SYNIVVAR, Syntax: illegal variable
SYNLABEL, Syntax: label expected
SYNLBRAC, Syntax: "[" expected

SYNLPAREN, Syntax: "(" expected
SYNLPASEM, Syntax: "(" or ";" expected
SYNLPCORB, Syntax: "(", "," or "]" expected
SYNLPSECO, Syntax: "(", ";" or ":" expected
SYNMECHEXPR, Syntax: mechanism specifier or expression expected
SYNNEWSTMT, Syntax: start of new statement expected
SYNOF, Syntax: OF expected
SYNPARMLST, Syntax: actual parameter list

SYNPARMSEC, Syntax: parameter section expected
SYNPERIOD Syntax: "." expected.
SYNPROMOD, Syntax: PROGRAM or MODULE expected
SYNQUOSTR, Syntax: quoted string expected
SYNRBRAC, Syntax: "]" expected
SYNRESWRD, Syntax: reserved word cannot be redefined
SYNRPAREN, Syntax: ")" expected
SYNRPASEM, Syntax: ";" or ")" expected
SYNRTNTYPCNF, Syntax: routine heading, type identifier or conformant parameter expected

SYNSEMI, Syntax: ";" expected
SYNSEMIEND, Syntax: ";" or END expected
SYNSEMMODI, Syntax: ";", "::", "^", or "[" expected
SYNSEMRB, Syntax: ";" or "]" expected
SYNSEOTEN, Syntax: ";", OTHERWISE or END expected
SYNTHEN, Syntax: THEN expected
SYNTODOWN, Syntax: TO or DOWNTO expected
SYNSEOTRP, Syntax: ";", OTHERWISE, or ")" expected
SYNTYPCNF, Syntax: type identifier or conformant parameter expected

SYNTYPID, Syntax: type identifier expected

Error: The compiler either failed to find an important lexical or syntactical element where one was expected, or it detected an error in such an element that does exist in your program.

SYNTYPPACK, Only ARRAY, FILE, RECORD or SET types can be PACKED

Warning: You cannot pack any type other than the structured types listed in the message.

SYNTYPSPEC, Syntax: type specification expected
SYNUNEXDECL, Syntax: declaration encountered in executable section
SYNUNTIL, Syntax: UNTIL expected
SYNXTRASEMI, Syntax: "; ELSE" is not valid Pascal, ELSE matched with IF on line "line number"

Error: The compiler either detected an error in a lexical or syntactical element in your program, or it failed to find such an element where one was expected.

TAGNOTORD, Tag type must be an ordinal type

Error: The type of a variant record's tag field must be one of the ordinal types.

TOOIDXEXPR, Too many index expressions; type has only "number of dimensions" dimensions

Error: A call to the UPPER or LOWER function specified an index value that exceeds the number of dimensions in the dynamic array.

TOPROGRAM, TO BEGIN/END DO not allowed in PROGRAM

Error: TO BEGIN DO and TO END DO declarations are only allowed in modules.

TYPCNTDISCR, Type can not be discriminated in this context

Error.

TYPFILSIZ, Type contains one or more FILE components, size attribute is illegal

Error: The allocation size of a FILE type cannot be controlled by a size attribute; therefore, you cannot use a size attribute on any type that has a file component.

TYPHASFILE, Type contains one or more FILE components

Error: Many operations are illegal on objects of type FILE and objects of structured types with file components; for example, you cannot initialize them, use them as value parameters, or read them with input procedures.

TYPHASNOVRNT, Type contains no variant part

Error: The formats of the NEW, DISPOSE, and SIZE routines that allow case labels to be specified can be used only when their parameters have variants.

TYPPTRFIL, Type must be pointer or FILE

Error: You cannot use the syntax "Variable^" to refer to an object whose type is not pointer or FILE.

TYPREF, %REF not allowed for this type

Error: The %REF foreign mechanism specifier cannot be used with schematic variables.

TYPSTDESCR, %STDESCR not allowed for this type

Error: The %STDESCR mechanism specifier is allowed only on objects of type CHAR, PACKED ARRAY [1..n] OF CHAR, VARYING OF CHAR, and arrays of these types.

TYPVARYCHR, Component type of VARYING must be CHAR

Error.

UNALIGNED, "variable name" is UNALIGNED

Error or Warning: You cannot use the data items listed in a call to the ADDRESS function, nor can you pass them as writable VAR, %REF, %DESCR, or %STDESCR parameters. This message is at warning level if the variable or component has the UNALIGNED attribute, and at error level if the variable or component is actually unaligned.

UNBPNTRET, "routine name" is not UNBOUND—only 32-bit address of entry point returned

Warning: The IADDRESS function returns an address as an integer. If you pass it the name of a routine, IADDRESS returns only the first 32 bits of the routine's bound procedure value.

UNCERTAIN, "Variable name" has not been initialized

Information.

UNDECLFRML, Undeclared formal parameter "symbol name"

Error: A formal parameter name listed in a nonpositional call to a routine does not match any of the formal parameters declared in the routine heading.

UNDECLID, Undeclared identifier "symbol name"

Error: In Pascal, an identifier must be declared before it is used. There are no default or implied declarations.

UNDSCHILL, Undiscriminated schema type is illegal

Error: An undiscriminated schema type does not have any actual discriminants. Without discriminants, the type size, any nested ARRAY bounds, and the offset of any nested RECORD fields are unknown.

UNINIT, "Variable name" may not have been initialized

Warning.

UNPREDRES, Calling FUNCTION "function name" declared FORWARD may yield unpredictable results

Warning: By using FORWARD declared functions in actual discriminant expressions, you can cause infinite loops at run-time or access violations.

UNREAD, Variable, "variable name" is assigned into, but never read

Information.

UNSCNFVRY, UNSAFE attribute not allowed on conformant VARYING parameter

Error.

UNSEXCRNG, UNSIGNED constant exceeds range

Error: The largest value allowed for an UNSIGNED integer is 4,294,967,295.

UNUSED, Variable, "variable name" is never referenced

Information.

UNWRITTEN, Variable "variable name" is read, but never assigned into

Warning.

UPLEVELACC, Unbound "routine name" precludes uplevel access to "variable name"

Error: A routine that was declared with the UNBOUND attribute cannot refer to automatic variables, routines, or labels declared in outer blocks.

UPLEVELGOTO, Unbound "routine name" precludes uplevel GOTO to "label name"

Error: A routine that was declared with the UNBOUND attribute cannot refer to automatic variables, routines, or labels declared in outer blocks.

USEDBFDECL, "symbol name" was used before being declared

Warning.

V1DYNARR, Decommitted Version 1 dynamic array type

Warning: The type syntax used to define a dynamic array parameter has been decommitted for the current version of VAX Pascal. You should edit your program to make the type definition conform to the current version conformant array syntax.

V1DYNARRASN, Decommitted Version 1 dynamic array assignment

Warning: In Version 1.0, dynamic arrays used in assignments could not be checked for compatibility until run time. This warning indicates that your program depends on an obsolete feature, which you should consider changing to reflect the current version syntax for conformant array parameters.

V1MISSPARM, Decommitted missing parameter syntax: correct by adding "number of commas" comma(s)

Warning: An OPEN procedure called with the decommitted Version 1.0 syntax fails to mark omitted parameters with commas. Your program depends on this obsolete feature, and you should insert the correct number of commas as listed in the message.

V1PARMSYN, Use of unsupported V1 omitted parameter syntax with new V2 feature(s)

Error: In a parameter list for the OPEN procedure, you cannot use both the Version 1.0 syntax for OPEN and the parameters that are new to subsequent versions of VAX Pascal.

V1RADIX, Decommitted Version 1 radix output specification

Information: In Version 1.0, octal and hexadecimal values could be written by placing the keywords OCT or HEX after a field width expression. Your program uses this obsolete feature; you should consider changing it to use the current versions OCT or HEX predeclared functions.

VALOUTBND, Value to be assigned is out of bounds

Error: A value specified in an array or record constructor exceeds the subrange defined as the type of the corresponding component.

VALUEINIT, VALUE variables must be initialized

Error: Variables with both the VALUE and GLOBAL attributes must be given an initial value in either the VAR section or in the VALUE section.

VALUETOOBIG, VALUE attribute not allowed on objects larger than 32 bits

Error: Variables with the VALUE attribute cannot be larger than 32 bits because they are expressed to the linker as global symbol references.

VALUETYP, VALUE allowed only on ordinal or real types

Error.

VALUEVISIB, GLOBAL or EXTERNAL visibility is required with the VALUE attribute

Error: Variables with the VALUE attribute must be given either external or global visibility. (If the variable is given global visibility, then it must also be given an initial value.)

VARCOMFRML, Variable is not compatible with formal parameter "formal parameter name"

Error: A variable being passed as an actual parameter is not compatible with the corresponding formal parameter indicated. Variable parameters must be structurally compatible. The reason for the incompatibility is provided in an informational message that the compiler prints along with this error message.

VARNOTEXT, Variable must be of type TEXT

Error: The EOLN function requires that its parameter be a file of type TEXT.

VARPRMRTN, Formal VAR parameter may not be a routine

Error: The reserved word VAR cannot precede the word PROCEDURE or FUNCTION in a formal parameter declaration.

VARPTRTYP, Variable must be of a pointer type

Error: The NEW and DISPOSE procedures operate only on pointer variables.

VARYFLDS, LENGTH and BODY are the only fields in a VARYING type

Error: You cannot use the syntax "Variable.Identifier" to specify any fields of a VARYING OF CHAR variable other than LENGTH and BODY.

VISAUTOCON, Visibility / AUTOMATIC allocation conflict

Error: The GLOBAL, EXTERNAL, WEAK_GLOBAL, and WEAK_EXTERNAL attributes require static allocation and therefore conflict with the AUTOMATIC attribute.

VISGLOBEXT, Visibilities are not GLOBAL/EXTERNAL or EXTERNAL/EXTERNAL

Information: In repeated declarations of a variable or routine, only one declaration at most can be global; all others must be external. This message can appear as additional information for other error messages.

VRNTRNG, Variant labels do not cover the range of the tag type

Error: According to the Pascal standard, you must specify one case label for each value in the tag type of a variant record or include an OTHERWISE clause.

WDTHONREAL, Second field width is allowed only when value is of a real type

Error: The fraction value in a field-width specification is allowed only for real-number values.

WRITEONLY, "variable name" is WRITEONLY

Warning: You cannot use a write-only variable in any context that requires the variable to be evaluated. For example, a write-only variable cannot be used as the control variable of a FOR statement.

XTRAERRORS, Additional diagnostics occurred on this line

Information: The number of errors occurring on this line exceeds the implementation's limit for outputting errors. You should correct the errors given and recompile your program.

ZERNOTALL, ZERO is not allowed for type or types containing "type name"

Error: ZERO may not be used to initialize objects of type FILE, TEXT, or TIMESTAMP or objects containing these types.

B.2 Run-Time Diagnostics

During execution, an image can generate a fatal error called an exception condition. When the VAX Pascal run-time system detects such a condition, the system displays an error message and aborts program execution. Run-time errors can also be issued by other facilities, such as the VMS Sort Utility or the VMS operating system.

VAX Pascal run-time system diagnostics are preceded by the following:

```
%PAS- F-
```

The severity level of a run-time error is F, fatal error.

Some conditions, particularly I/O errors, may cause several messages to be generated. The first message is a diagnostic that specifies the file that was being accessed (if any) when the error occurred and the nature of the error. Next, an RMS error message may be generated. In most cases, you should be able to understand the error by looking up the first message in the following list. If not, see the *VMS System Messages and Recovery Procedures Reference Manual* for an explanation of the RMS error message.

All diagnostic messages contain a brief explanation of the event that caused the error. This section lists run-time diagnostic messages in alphabetical order, including explanatory message text. Where the message text is not self-explanatory, additional explanation follows. Portions of the message text enclosed in quotation marks are items that the compiler substitutes with the name of a data object when it generates the message.

ACCMETINC, ACCESS_METHOD specified is incompatible with this file

Explanation: The value of the ACCESS_METHOD parameter for a call to the OPEN procedure is not compatible with the file's organization or record type. You can use DIRECT access only with files that have relative organization or sequential organization and fixed-length records. You can use KEYED access only with indexed files.

User Action: Make sure that you are accessing the correct file. See Chapter 5 to determine which access method you should use.

AMBVALENU, "string" is an ambiguous value for enumerated type "type"

Explanation: While a value of an enumerated type was being read from a text file, not enough characters of the identifier were found to specify an unambiguous value.

User Action: Specify enough characters of the identifier so that it is not ambiguous.

ARRINDVAL, array index value is out of range

Explanation: You enabled bounds checking for a compilation unit and attempted to specify an index that is outside the array's index bounds.

User Action: Correct the program or data so that all references to array indexes are within the declared bounds.

ARRNOTCOM, conformant array is not compatible

Explanation: You attempted to assign one dynamic array to another that did not have the same index bounds. This error occurs only when the arrays use the decommitted Version 1.0 syntax for dynamic array parameters.

User Action: Correct the program so that the two dynamic arrays have the same index bounds. You could also change the arrays to conform to the current syntax for conformant arrays; most incompatibilities could then be detected at compile time rather than at run time. See the *VAX Pascal Reference Manual* for more information on current conformant arrays.

ARRNOTSTR, conformant array is not a string

Explanation: In a string operation, you used a conformant PACKED ARRAY OF CHAR value whose index had a lower bound not equal to 1 or an upper bound greater than 65535.

User Action: Correct the array's index so that the array is a character string.

BUGCHECK, internal consistency failure "nnn" in Pascal Run-Time Library

Explanation: The VAX Pascal Run-Time Library has detected an internal error or inconsistency. This problem may be caused by an out-of-bounds array reference or a similar error in your program.

User Action: Rerun your program with all CHECK options enabled. If you are unable to find an error in your program, please submit a Software Performance Report (SPR) to Digital, including a machine-readable copy of your program, data, and a sample execution illustrating the problem.

BADBITARG, Nbits argument to DEC or UDEC must be between 1 and 32

Explanation: The value being passed to DEC or UDEC exceeds the boundaries of a longword (32 bits).

User Action: If you are calling PAS$DEC or PAS$UDEC directly, examine your program for bad data. If PAS$DEC or PAS$UDEC have been called through predeclared routines in your program, then submit a Software Performance Report (SPR).

CANCNTERR, handler cannot continue from a nonfile error

Explanation: A user condition handler attempted to return SS$_CONTINUE for an error not involving file input/output. To recover from such an error, you must use either an uplevel GOTO statement or the SYS$UNWIND system service.

User Action: Modify the user handler to use one of the allowed recovery actions for nonfile errors, or to resignal the error if no recovery action is possible.

CASSELVAL, CASE selector value is out of range

Explanation: The value of the case selector in a CASE statement does not equal any of the specified case labels, and the statement has no OTHERWISE clause.

User Action: Either add an OTHERWISE clause to the CASE statement or change the value of the case selector so that it equals one of the case labels. See the *VAX Pascal Reference Manual* for more information.

CONCATLEN, string concatenation has more than 65535 characters

Explanation: The result of a string concatenation operation would result in a string longer than 65,535 characters, which is the maximum length of a string.

User Action: Correct the program so that all concatenations result in strings no longer than 65,535 characters.

CSTRCOMISS, invalid constructor: component(s) missing

Explanation: The constructor did not specify sufficient component values to initialize a variable of the type.

User Action: Specify more components in the constructor, use the OTHERWISE clause in the constructor, or modify the type definition to specify fewer components.

CURCOMUND, current component is undefined for DELETE or UPDATE

Explanation: You attempted a DELETE or UPDATE procedure when no current component was defined. A current component is defined by a successful GET, FIND, FINDK, RESET, or RESETK that locks the component. Files opened with HISTORY:=READONLY never lock components.

User Action: Correct the program so that a current component is defined before executing DELETE or UPDATE.

DELNOTALL, DELETE is not allowed for a sequential organization file

Explanation: You attempted a DELETE procedure for a file with sequential organization, which is not allowed. DELETE is valid only on files with relative or indexed organization.

User Action: Make sure that the program is referencing the correct file. See Chapter 5 to determine what file characteristics are appropriate for your application.

ERRDURCLO, error during CLOSE

Explanation: RMS reported an unexpected error during execution of the CLOSE procedure. The RMS error message is also displayed. This message may also be issued with error severity when files are implicitly closed during a procedure or image exit.

User Action: See the *VMS System Messages and Recovery Procedures Reference Manual* for the description of the RMS error.

ERRDURDEL, error during DELETE

Explanation: RMS reported an unexpected error during execution of a DELETE procedure. The RMS error message is also displayed.

User Action: See the *VMS System Messages and Recovery Procedures Reference Manual* for the description of the RMS error.

ERRDURDIS, error during DISPOSE

Explanation: An error occurred during execution of a DISPOSE procedure. An additional message that further describes the error may also be displayed.

User Action: Make sure that the heap storage being freed was allocated by a successful call to the NEW procedure, and that it has not been already freed. If an additional message is shown, see the *VMS System Messages and Recovery Procedures Reference Manual* for the description of that message.

ERRDUREXT, error during EXTEND

Explanation: RMS reported an unexpected error during execution of an EXTEND procedure. The RMS error message is also displayed.

User Action: See the *VMS System Messages and Recovery Procedures Reference Manual* for the description of the RMS error.

ERRDURFIN, error during FIND or FINDK

Explanation: RMS reported an unexpected error during execution of a FIND or FINDK procedure. The RMS error message is also displayed.

User Action: See the *VMS System Messages and Recovery Procedures Reference Manual* for the description of the RMS error.

ERRDURGET, error during GET

Explanation: RMS reported an unexpected error during execution of the GET procedure. The RMS error message is also displayed.

User Action: See the *VMS System Messages and Recovery Procedures Reference Manual* for the description of the RMS error.

ERRDURMAR, error during MARK

Explanation: An error occurred during execution of the PAS$MARK2 procedure. An additional message is displayed that further describes the error.

User Action: See the *VMS System Messages and Recovery Procedures Reference Manual* for a description of the additional message.

ERRDURNEW, error during NEW

Explanation: An error occurred during execution of the NEW procedure. An additional message is displayed that further describes the error.

User Action: See the *VMS System Messages and Recovery Procedures Reference Manual* for a description of the additional message.

ERRDUROPE, error during OPEN

Explanation: An unexpected error occurred during execution of the OPEN procedure, or during an implicit open caused by a RESET or REWRITE procedure. An additional message is displayed that further describes the error.

User Action: See the *VMS System Messages and Recovery Procedures Reference Manual* for a description of the additional message.

ERRDURPRO, error during prompting

Explanation: RMS reported an unexpected error during output of partial lines to a terminal. The RMS error message is also displayed.

User Action: See the *VMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

ERRDURPUT, error during PUT

Explanation: RMS reported an unexpected error during execution of the PUT procedure. The RMS error message is also displayed.

User Action: See the *VMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS message.

ERRDURREL, error during RELEASE

Explanation: An unexpected error occurred during execution of the PAS$RELEASE2 procedure. An additional message may be displayed that further describes the error.

User Action: Make sure that the marker argument was returned from a successful call to PAS$MARK2 and that the storage has not been already freed. If an additional message is displayed, see the *VMS System Messages and Recovery Procedures Reference Manual* for a description of that message.

ERRDURRES, error during RESET or RESETK

Explanation: RMS reported an unexpected error during execution of the RESET or RESETK procedure. The RMS error message is also displayed.

User Action: See the *VMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

ERRDURREW, error during REWRITE

Explanation: RMS reported an unexpected error during execution of the REWRITE procedure. The RMS error message is also displayed.

User Action: See the *VMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

ERRDURTRU, error during TRUNCATE

Explanation: RMS reported an unexpected error during execution of the TRUNCATE procedure. The RMS error message is also displayed.

User Action: See the *VMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

ERRDURUNL, error during UNLOCK

Explanation: RMS reported an unexpected error during execution of the UNLOCK procedure. The RMS error message is also displayed.

User Action: See the *VMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

ERRDURUPD, error during UPDATE

Explanation: RMS reported an unexpected error during execution of the UPDATE procedure. The RMS error message is also displayed.

User Action: See the *VMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

ERRDURWRI, error during WRITELN

Explanation: RMS reported an unexpected error during execution of the WRITELN procedure. The RMS error message is also displayed.

User Action: See the *VMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

EXTNOTALL, EXTEND is not allowed for a shared file

Explanation: Your program attempted an EXTEND procedure for a file for which the program did not have exclusive access. EXTEND requires that no other users be allowed to access the file. Note that this message may also be issued if you do not have permission to extend to the file.

User Action: Correct the program so that the file is opened with SHARING:=NONE, which is the default, before performing an EXTEND procedure.

FAIGETLOC, failed to GET locked component

Explanation: Your program attempted to access a component of a file that was locked by another user. You can usually expect this condition to occur when more than one user is accessing the same relative or indexed file.

User Action: Determine whether this condition should be allowed to occur. If so, modify your program so that it detects the condition and retries the operation later. See Chapter 5 for more information.

FILALRACT, file "file name" is already active

Explanation: Your program attempted a file operation on a file for which another operation was still in progress. This error can occur if a file is used in AST or condition-handling routines.

User Action: Modify your program so that it does not try to use files that may currently be in use.

FILALRCLO, file is already closed

Explanation: Your program attempted to close a file that was already closed.

User Action: Modify your program so that it does not try to close files that are not open.

FILALROPE, file is already open

Explanation: Your program attempted to open a file that was already open.

User Action: Modify your program so that it does not try to open files that are already open.

FILNAMREQ, FILE_NAME required for this HISTORY or DISPOSITION

Explanation: Your program attempted to open a nonexternal file without specifying a file name parameter to the OPEN procedure, but the HISTORY or DISPOSITION parameter specified requires a file name.

User Action: Add a file name parameter to the OPEN procedure call, specifying an appropriate file name.

FILNOTDIR, file is not opened for direct access

Explanation: Your program attempted to execute a DELETE, FIND, LOCATE, or UPDATE procedure on a file that was not opened for direct access.

User Action: Modify the program to specify the ACCESS_METHOD:=DIRECT parameter to the OPEN procedure when opening the file. See Chapter 5 to determine if direct access is appropriate for your application.

FILNOTFOU, file not found

Explanation: Your program attempted to open a file that does not exist. An additional RMS message is displayed that further describes the problem.

User Action: Make sure that you are specifying the correct file. See the *VMS System Messages and Recovery Procedures Reference Manual* for a description of the additional RMS message.

FILNOTGEN, file is not in Generation mode

Explanation: Your program attempted a file operation that required the file to be in generation mode (ready for writing).

User Action: Modify the program to use a REWRITE, TRUNCATE, or LOCATE procedure to place the file in generation mode as appropriate. See Chapter 5 for more information.

FILNOTINS, file is not in Inspection mode

Explanation: Your program attempted a file operation that required the file to be in inspection mode (ready for reading).

User Action: Modify the program to use a RESET, RESETK, FIND, or FINDK procedure to place the file in inspection mode as appropriate. See Chapter 5 for more information.

FILNOTKEY, file is not opened for keyed access

Explanation: Your program attempted to execute a FINDK, RESETK, DELETE, or UPDATE procedure on a file that was not opened for keyed access.

User Action: Modify the program to specify the ACCESS_METHOD:=KEYED parameter to the OPEN procedure when opening the file. See Chapter 5 to make sure that keyed access is appropriate to your application.

FILNOTOPE, file is not open

Explanation: Your program attempted to execute a file manipulation procedure on a file that was not open.

User Action: Correct the program to open the file using a RESET, REWRITE, or OPEN procedure as appropriate. See Chapter 5 for more information.

FILNOTSEQ, file is not sequential organization

Explanation: Your program attempted to execute the TRUNCATE procedure on a file that does not have sequential organization. TRUNCATE is valid only on sequential files.

User Action: Make sure that your program is accessing the correct file. Correct the program so that all TRUNCATE operations are performed on sequential files.

FILNOTTEX, file is not a textfile

Explanation: Your program performed a file operation that required a file of type TEXT on a nontext file. Note that the type FILE OF CHAR is not equivalent to TEXT unless you have compiled the program with the /OLD_VERSION qualifier.

User Action: Make sure that your program is accessing the correct file. Correct the program so that a text file is always used when required.

GENNOTALL, Generation mode is not allowed for a READONLY file

Explanation: Your program attempted to place a file declared with the READONLY attribute into generation mode, which is not allowed. Note that the READONLY file attribute is not equivalent to the HISTORY:=READONLY parameter to the OPEN procedure.

User Action: Correct the program so that the file either does not have the READONLY attribute or is not placed into generation mode.

GETAFTEOF, GET attempted after end-of-file

Explanation: Your program attempted a GET operation on a file while EOF(f) was TRUE. This situation occurs when a previous GET operation (possibly implicitly performed by a RESET, RESETK, or READ procedure) reads to the end of the file and causes the EOF(f) function to return TRUE. If another GET is then performed, this error is given.

User Action: Correct the program so that it either tests whether EOF(f) is TRUE, before attempting a GET operation, or repositions the file before the end-of-file marker.

GOTOFAILED, non-local GOTO failed

Explanation: An error occurred while a nonlocal GOTO statement was being executed. This error might occur because of an error in the user program, such as an out-of-bounds array reference.

User Action: Rerun your program, enabling all CHECK options. If you cannot locate an error in your program and the problem persists, please submit a Software Performance Report (SPR) to Digital, and include a machine-readable copy of your program, data, and results of a sample execution illustrating the problem.

HALT, HALT procedure called

Explanation: The program terminated its execution by executing the HALT procedure. This message is solely informational.

User Action: None.

ILLGOTO, illegal uplevel GOTO during routine activation

Explanation: An uplevel GOTO was made into the body of a routine before the declaration part of the routine was completely processed.

User Action: Correct the program to avoid the uplevel GOTO until the declaration part has been completely processed.

INSNOTALL, Inspection mode is not allowed for a WRITEONLY file

Explanation: Your program attempted to place a file declared with the WRITEONLY attribute into inspection mode, which is not allowed.

User Action: Correct the program so that the file variable either does not have the WRITEONLY attribute or is not placed into inspection mode.

INSVIRMEM, insufficient virtual memory

Explanation: The VMS Run-Time Library was unable to allocate enough heap storage to open the file.

User Action: Examine your program to see whether it is making excessive use of heap storage, which might be allocated using the NEW procedure or the Run-Time Library procedure LIB$GET_VM. Modify your program to free any heap storage it does not need.

INVARGPAS, invalid argument to Pascal Run-Time Library

Explanation: An invalid argument or inconsistent data structure was passed to the VMS Run-Time Library by the compiled code, or a system service returned an unrecognized value to the Run-Time Library.

User Action: Rerun your program with all CHECK options enabled. Make sure that the version of the current operating system is compatible with the version of the compiler. If you cannot locate an error in your program and the problem persists, please submit a Software Performance Report (SPR) to Digital, and include a machine-readable copy of your program, data, and results of a sample execution illustrating the problem.

INVFILSYN, invalid file name syntax

Explanation: Your program attempted to open a file with an invalid file name. The file name used can be derived from the file variable name, the value of the file name parameter to the OPEN procedure, or the logical name translations (if any) of the file variable name and portions of the file name parameter and your default device and directory. The displayed text may include the erroneous file name. This error can also occur if the value of the file name parameter is longer than 255 characters. Additional RMS messages may be displayed that further describe the error.

User Action: Use the information provided in the displayed messages to determine which component of the file name is invalid. Verify that any logical names used are defined correctly. See the *VAX Pascal Reference Manual* for information on file names.

INVFILVAR, invalid file variable at location "nnn"

Explanation: The file variable passed to a VMS Run-Time Library procedure was invalid or corrupted. This problem might be caused by an error in the user program, such as an out-of-bounds array access. It can also occur if a file variable is passed from a routine compiled with a version of VAX Pascal earlier than Version 2.0 to a routine compiled with a later version of the compiler, or if the new key options are used on VMS systems earlier than Version 4.6.

User Action: Rerun your program with all CHECK options enabled, and recompile all modules using the same compiler. If the problem persists, please submit a Software Performance Report (SPR) to Digital, and include a machine-readable copy of your

program, data, and results of a sample execution illustrating the problem.

INVKEYDEF, invalid key definition

Explanation: Your program attempted to open a file of type RECORD whose component type contained a field with an invalid KEY attribute. One of the following errors occurred:

- A new file was being created and the key numbers were not dense.
- A key field was defined at an offset of more than 65,535 bytes from the beginning of the record.

User Action: If a new file is being created, make sure that the key fields are numbered consecutively, starting with 0 for the required primary key. If you are opening an existing file, you must explicitly specify HISTORY:=OLD or HISTORY:=READONLY as a parameter to the OPEN procedure. Make sure that the length of the record is within the maximum permitted for the file organization being used. See Chapter 5 for more information.

INVRECLEN, invalid record length of "nnn"

Explanation: A file was being opened, and one of the following errors occurred:

- The length of the file components was greater than that allowed for the file organization and record format (for most operations, the largest length allowed is 32,765 bytes).
- The value of the RECORD_LENGTH parameter to the OPEN procedure was greater than that allowed for the file organization and record format (for most operations, the largest value allowed is 32,765 bytes).

User Action: Correct the program so that the record length used is within the permitted limits for the type of file being used. See *VAX Record Management Services Manual* for more information.

INVSYNBIN, "string" is invalid syntax for a binary value

Explanation: While a READ or READV procedure was reading a binary value from a text file, the characters read did not conform to the syntax for a binary value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action: Correct the program or the input data so that the correct syntax is used. See the *VAX Pascal Reference Manual* for more information.

INVSYNHEX, "string" is invalid syntax for a hexadecimal value

Explanation: While a READ or READV procedure was reading a hexadecimal value from a text file, the characters read did not conform to the syntax for an hexadecimal value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action: Correct the program or the input data so that the correct syntax is used. See the *VAX Pascal Reference Manual* for more information.

INVSYNENU, "string" is invalid syntax for an enumerated value

Explanation: While a READ or READV procedure was reading an identifier of an enumerated type from a text file, the characters read did not conform to the syntax for an enumerated value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action: Correct the program or the input data so that the correct syntax is used. See the *VAX Pascal Reference Manual* for more information.

INVSYNINT, "string" is invalid syntax for an integer value

Explanation: While a READ or READV procedure was reading a value for an integer identifier from a text file, the characters read did not conform to the syntax for an integer value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action: Correct the program or the input data so that the correct syntax is used. See the *VAX Pascal Reference Manual* for more information.

INVSYNOCT, "string" is invalid syntax for an octal value

Explanation: While a READ or READV procedure was reading an octal value from a text file, the characters read did not conform to the syntax for an octal value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action: Correct the program or the input data so that the correct syntax is used. See the *VAX Pascal Reference Manual* for more information.

INVSYNREA, "string" is invalid syntax for a real value

Explanation: While a READ or READV procedure was reading a value for a real identifier from a text file, the characters read did not conform to the syntax for a real value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action: Correct the program or the input data so that the correct syntax is used. See the *VAX Pascal Reference Manual* for more information.

INVSYNUNS, "string" is invalid syntax for an unsigned value

Explanation: While a READ or READV procedure was reading a value for an unsigned identifier from a text file, the characters read did not conform to the syntax for an unsigned value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action: Correct the program or the input data so that the correct syntax is used. See the *VAX Pascal Reference Manual* for more information.

KEYCHANOT, key field change is not allowed

Explanation: Your program attempted an UPDATE procedure for a record of an indexed file that would have changed the value of a key field, and this situation was disallowed when the file was created.

User Action: If the program needs to detect this situation when it occurs, specify the ERROR:=CONTINUE parameter for the UPDATE procedure, and use the STATUS function to determine which error, if any, occurred. If necessary, modify the program so that it does not improperly change a key field, or recreate the file

specifying that the key field is permitted to change. See Chapter 5 for more information.

KEYDEFINC, KEY "nnn" definition is inconsistent with this file

Explanation: An indexed file of type RECORD was opened, and the component type contained fields whose KEY attributes did not match those of the existing file. The number of the key in error is displayed in the message.

User Action: Correct the RECORD definition so that it describes the correct KEY fields, or recreate the file so that it matches the declared keys. See Chapter 5 for more information.

KEYDUPNOT, key field duplication is not allowed

Explanation: Your program attempted an UPDATE or PUT procedure for a record of an indexed file that would have duplicated a key field value of an existing record, and this situation was disallowed when the file was created.

User Action: If the program needs to detect this situation when it occurs, specify the ERROR:=CONTINUE parameter for the PUT or UPDATE procedure, and use the STATUS function to determine which error, if any, occurred. If necessary, modify the program so that it does not improperly duplicate a key field, or recreate the file specifying that the key field is permitted to be duplicated. See Chapter 5 for more information.

KEYNOTDEF, KEY "nnn" is not defined for this file

Explanation: Your program attempted a FINDK or RESETK procedure on an indexed file, and the key number specified does not exist in the file.

User Action: Correct the program so that the correct key numbers are used when accessing the file.

KEYVALINC, key value is incompatible with the file's key "nnn"

Explanation: The key value specified for the FINDK procedure was incompatible in type or size with the key field of the file, or your program attempted an OPEN on an existing file and the key check failed.

User Action: Make sure that the correct key value is being specified for FINDK and OPEN. Correct the program so that the type of the key value is compatible with the key of the file. See the *VAX Pascal Reference Manual* for more information.

LINTOOLON, line is too long, exceeded record length by "nnn" character(s)

Explanation: Your program attempted a WRITE, PUT, WRITEV, or other output procedure on a text file that would have placed more characters in the current line than the record length of the file would allow. The number of characters that did not fit is displayed in the message.

User Action: Correct the program so that it does not place too many characters in the current line. If appropriate, use the WRITELN procedure, or specify an increased record length parameter when opening the file with the OPEN procedure.

LINVALEXC, LINELIMIT value exceeded

Explanation: The number of lines written to the file exceeded the maximum specified as the line limit. The line limit value is determined by the translation of the logical name PAS$LINELIMIT, if any, or the value specified in a call to the LINELIMIT procedure for the file.

User Action: As appropriate, correct the program so that it does not write as many lines, or increase the line limit for the file. Note that if a line limit is specified for a nontext file, each PUT procedure called for the file is considered to be one line. See the *VAX Pascal Reference Manual* for more information.

LOWGTRHIGH, low-bound exceeds high-bound

Explanation: The lower bound of a subrange definition is larger than the higher bound.

User Action: Modify the declaration so the lower bound is less than or equal to the higher bound.

MAXLENRNG, maximum length must be in range 1..65535

Explanation: The maximum length for a string type is 65,535.

User Action: Modify the declaration to specify a smaller amount.

MODNEGNUM, MOD of a negative modulus has no mathematical definition

Explanation: In the MOD operation A MOD B, the operand B must have a positive integer value.

User Action: Correct the program so that the operand B has a positive integer value.

NEGDIGARG, negative Digits argument to BIN, HEX or OCT is not allowed

Explanation: Your program attempted to specify a negative value for the Digits argument in a call to the BIN, HEX, or OCT procedure, which is not permitted.

User Action: Correct the program so that only nonnegative Digits arguments are used for calls to BIN, HEX, and OCT.

NEGWIDDIG, negative Width or Digits specification is not allowed

Explanation: A WRITE or WRITEV procedure on a text file contained a field width specification that included a negative Width or Digits value, which is not permitted.

User Action: Correct the program so that only nonnegative Width and Digits parameters are used.

NOTVALTYP, "string" is not a value of type "type"

Explanation: Your program attempted a READ or READV procedure on a text file, but the value read could not be expressed in the specified type. For example, this error results if a real value read is outside the range of the identifier's type, or if an enumerated value is read that does not match any of the valid constant identifiers in its type.

User Action: Correct the program or the input data so that the values read are compatible with the types of the identifiers receiving the data.

OPNDASSCOM, operands are not assignment compatible

Explanation: The operands do not have the same type.

User Action: Examine the declarations of the operands and make sure they have compatible types.

ORDVALOUT, ordinal value is out of range

Explanation: A value of an ordinal type is outside the range of values specified by the type. For example, this error results if you try to use the SUCC function on the last value in the type or the PRED function on the first value.

User Action: Correct the program so that all ordinal values are within the range of values specified by the ordinal type.

ORGSPEINC, ORGANIZATION specified is inconsistent with this file

Explanation: The value of the ORGANIZATION parameter for the OPEN procedure that opened an existing file was inconsistent with the actual organization of the file.

User Action: Correct the program so that the correct organization is specified. See Chapter 5 for more information.

PADLENERR, PAD length error

Explanation: The length of the character string to be padded by the PAD function is greater than the length specified as the finished size, or the finished size specified is greater than 65,535.

User Action: Correct the call to PAD so that the finished size specified describes a character string of the correct length. See the *VAX Pascal Reference Manual* for the rules governing the PAD function.

PTRREFNIL, pointer reference to NIL

Explanation: Your program attempted to evaluate a pointer value while its value was NIL.

User Action: Make sure that the pointer has a value before you try to evaluate it. See the *VAX Pascal Reference Manual* for more information on pointer values.

RECLENINC, RECORD_LENGTH specified is inconsistent with this file

Explanation: The record length obtained from the file component's length or from the value of the record length parameter specified for the OPEN procedure was inconsistent with the actual record length of an existing file.

User Action: Correct the program so that the record length specified, if any, is consistent with the file. See Chapter 5 for more information.

RECTYPINC, RECORD_TYPE specified is inconsistent with this file

Explanation: The value of the RECORD_LENGTH parameter specified for the OPEN procedure was inconsistent with the actual record type of an existing file.

User Action: Correct the program so that the record type specified, if any, is consistent with the file. See Chapter 5 for more information.

REFINAVAR, read or write of inactive variant

Explanation: A field of an inactive variant was read or written.

User Action: Correct the program so the variant is active or remove the reference to the inactive field.

RESNOTALL, RESET is not allowed on an unopened internal file

Explanation: Your program attempted a RESET procedure for a nonexternal file that was not open. This operation is not permitted because RESET must operate on an existing file, and there is no information associated with a nonexternal file that allows RESET to open it.

User Action: Correct the program so that nonexternal files are opened before using RESET. Either OPEN or REWRITE may be used to open a nonexternal file. See the *VAX Pascal Reference Manual* for more information.

REWNOTALL, REWRITE is not allowed for a shared file

Explanation: Your program attempted a REWRITE procedure for a file for which the program did not have exclusive access. REWRITE requires that no other users be allowed to access the file while the file's data is deleted. Note that this message may also be issued if you do not have permission to write to the file.

User Action: Correct the program so that the file is opened with SHARING := NONE, which is the default, before performing a REWRITE procedure.

SETASGVAL, set assignment value has element out of range

Explanation: Your program attempted to assign to a set variable a value that is outside the range specified by the variable's component type.

User Action: Correct the assignment statement so that the value being assigned falls within the component type of the set variable. See the *VAX Pascal Reference Manual* for more information on sets.

SETCONVAL, set constructor value out of range

Explanation: Your program attempted to include in a set constructor a value that is outside the range specified by the set's component type, or a value that is greater than 255 or less than 0.

User Action: Correct the constructor so that it includes only those values within the range of the set's component type. See the *VAX Pascal Reference Manual* for more information on sets.

SETNOTRNG, set element is not in range 0..255

Explanation: Sets of INTEGER or UNSIGNED must be in the range of 0..255.

User Action: Modify the declaration to specify a smaller range.

STRASGLEN, string assignment length error

Explanation: Your program attempted to assign to a string variable a character string that is longer than the declared maximum length of the variable (if the variable's type is VARYING) or that is not of the same length as the variable (if the variable's type is PACKED ARRAY OF CHAR).

User Action: Correct the program so that the string is of a correct length for the variable to which it is being assigned.

STRCOMLEN, string comparison length error

Explanation: Your program attempted to compare two character strings that do not have the same current length.

User Action: Correct the program so that the two strings have the same length at the time of the comparison.

SUBASGVAL, subrange assignment value out of range

Explanation: Your program attempted to assign to a subrange variable a value that is not contained in the subrange type.

User Action: Correct the program so that all values assigned to a subrange variable fall within the variable's type.

SUBSTRSEL, SUBSTR selection error

Explanation: A SUBSTR function attempted to extract a substring that was not entirely contained in the original string.

User Action: Correct the call to SUBSTR so that it specifies a substring that can be extracted from the original string. See the *VAX Pascal Reference Manual* for complete information on the SUBSTR function.

TEXREQSEQ, textfiles require sequential organization and access

Explanation: Your program attempted to open a file of type TEXT that either did not have sequential organization, or had an ACCESS_METHOD other than SEQUENTIAL (the default) when opened by the OPEN procedure.

User Action: Make sure that the program refers to the correct file. Correct the program so that only sequential organization and access are used for text files.

TRUNOTALL, TRUNCATE is not allowed for a shared file

Explanation: Your program attempted to call the TRUNCATE procedure for a file that was opened for shared access. You cannot truncate files that might be shared by other users. This message may also be issued if you do not have permission to write to the file.

User Action: Correct the program so that it does not try to truncate shared files. If the file is opened with the OPEN procedure, do not specify a value other than NONE (the default) for the SHARING parameter.

UPDNOTALL, UPDATE not allowed for a sequential organization file

Explanation: Your program attempted to call the UPDATE procedure for a sequential file. UPDATE is valid only on relative and indexed files.

User Action: Correct the program so that it does not try to use UPDATE for sequential files, or recreate the file with relative or indexed organization. If you are using direct access on a sequential file, individual records can be updated with the LOCATE and PUT procedures. See Chapter 5 to determine whether a different file organization may be appropriate for your application.

VARINDVAL, VARYING index value exceeds current length

Explanation: The index value specified for a VARYING OF CHAR string is greater than the string's current length.

User Action: Correct the index value so that it specifies a legal character in the string.

WRIINVENU, WRITE of an invalid enumerated value

Explanation: Your program attempted to write an enumerated value using a WRITE or WRITEV procedure, but the internal representation of that value was outside the possible range for the enumerated type.

User Action: Verify that your program is not improperly using PRED, SUCC, or type casting to assign an invalid value to a variable of enumerated type.

Appendix C

Errors Returned by STATUS and STATUSV Functions

This appendix lists the error conditions detected by the STATUS and STATUSV functions, their symbolic names, and the corresponding values. The symbolic names and their values are defined in the file SYS$LIBRARY:PASSTATUS.PAS, which you can include with a %INCLUDE directive in a CONST section of your program. To test for a specific condition, you compare the STATUS or STATUSV return values against the value of a symbolic name.

Note that the symbolic names correspond to some of the run-time errors listed in Appendix B; however, not all run-time errors can be detected by STATUS.

There is a one-to-one correspondence between the symbolic constants returned by STATUS or STATUSV documented in PASSTATUS and the VAX condition code values in SYS$LIBRARY:PASDEF.PAS. The following routine shows how to map the return value of STATUS to its corresponding condition code located in PASDEF.PAS:

```
FUNCTION CONVERT_STATUS_TO_CONDITION(STAT:INTEGER):INTEGER;
   BEGIN
      CONVERT_STATUS_TO_CONDITION := %x218644 + STAT * 8;
   END;
```

Table C–1 lists the symbolic names and the values returned by the STATUS and STATUSV functions and explains the error condition that corresponds to each value.

Table C–1: STATUS and STATUSV Return Values

Name	Value	Meaning
PAS$K_ACCMETINC	5	Specified access method is not compatible with this file.
PAS$K_AMBVALENU	30	"String" is an ambiguous value for the enumerated type "type".
PAS$K_CURCOMUND	73	DELETE or UPDATE was attempted while the current component was undefined.
PAS$K_DELNOTALL	100	DELETE is not allowed for a file with sequential organization.
PAS$K_EOF	–1	File is at end-of-file.
PAS$K_ERRDURCLO	16	Error occurred while the file was being closed.
PAS$K_ERRDURDEL	101	Error occurred during execution of DELETE.
PAS$K_ERRDUREXT	127	Error occurred during execution of EXTEND.
PAS$K_ERRDURFIN	102	Error occurred during execution of FIND or FINDK.
PAS$K_ERRDURGET	103	Error occurred during execution of GET.
PAS$K_ERRDUROPE	2	Error occurred during execution of OPEN.
PAS$K_ERRDURPRO	36	Error occurred during prompting.
PAS$K_ERRDURPUT	104	Error occurred during execution of PUT.
PAS$K_ERRDURRES	105	Error occurred during execution of RESET or RESETK.
PAS$K_ERRDURREW	106	Error occurred during execution of REWRITE.
PAS$K_ERRDURTRU	107	Error occurred during execution of TRUNCATE.
PAS$K_ERRDURUNL	108	Error occurred during execution of UNLOCK.
PAS$K_ERRDURUPD	109	Error occurred during execution of UPDATE.
PAS$K_ERRDURWRI	50	Error occurred during execution of WRITELN.
PAS$K_EXTNOTALL	128	EXTEND is not allowed for a shared file.
PAS$K_FAIGETLOC	74	GET failed to retrieve a locked component.
PAS$K_FILALRCLO	15	File is already closed.
PAS$K_FILALROPE	1	File is already open.

(continued on next page)

Table C–1 (Cont.): STATUS and STATUSV Return Values

Name	Value	Meaning
PAS$K_FILNAMREQ	14	File name must be specified in order to save, print, or submit an internal file.
PAS$K_FILNOTDIR	110	File is not open for direct access.
PAS$K_FILNOTFOU	3	File was not found.
PAS$K_FILNOTGEN	111	File is not in generation mode.
PAS$K_FILNOTINS	112	File is not in inspection mode.
PAS$K_FILNOTKEY	113	File is not open for keyed access.
PAS$K_FILNOTOPE	114	File is not open.
PAS$K_FILNOTSEQ	115	File does not have sequential organization.
PAS$K_FILNOTTEX	116	File is not a text file.
PAS$K_GENNOTALL	117	Generation mode is not allowed for a read-only file.
PAS$K_GETAFTEOF	118	GET attempted after end-of-file has been reached.
PAS$K_INSNOTALL	119	Inspection mode is not allowed for a write-only file.
PAS$K_INSVIRMEM	120	Insufficient virtual memory.
PAS$K_INVARGPAS	121	Invalid argument passed to a VAX Pascal Run-Time Library procedure.
PAS$K_INVFILSYN	4	Invalid syntax for file name.
PAS$K_INVKEYDEF	9	Key definition is invalid.
PAS$K_INVRECLEN	12	Record length nnn is invalid.
PAS$K_INVSYNBIN	37	"String" is invalid syntax for a binary value.
PAS$K_INVSYNENU	31	"String" is invalid syntax for a value of an enumerated type.
PAS$K_INVSYNHEX	38	"String" is invalid syntax for a hexadecimal value.
PAS$K_INVSYNINT	32	"String" is invalid syntax for an integer.
PAS$K_INVSYNOCT	39	"String" is invalid syntax for an octal value.
PAS$K_INVSYNREA	33	"String" is invalid syntax for a real number.

(continued on next page)

Table C–1 (Cont.): STATUS and STATUSV Return Values

Name	Value	Meaning
PAS$K_INVSYNUNS	34	"String" is invalid syntax for an unsigned integer.
PAS$K_KEYCHANOT	72	Changing the key field is not allowed.
PAS$K_KEYDEFINC	10	KEY(nnn) definition is inconsistent with this file.
PAS$K_KEYDUPNOT	71	Duplication of key field is not allowed.
PAS$K_KEYNOTDEF	11	KEY(nnn) is not defined in this file.
PAS$K_KEYVALINC	70	Key value is incompatible with file's key nnn.
PAS$K_LINTOOLON	52	Line is too long; exceeds record length by nnn characters.
PAS$K_LINVALEXC	122	LINELIMIT value exceeded.
PAS$K_NEGWIDDIG	53	Negative value in width or digits (of a field width specification) is invalid.
PAS$K_NOTVALTYP	35	"String" is not a value of type "type".
PAS$K_ORGSPEINC	8	Specified organization is inconsistent with this file.
PAS$K_RECLENINC	6	Specified record length is inconsistent with this file.
PAS$K_RECTYPINC	7	Specified record type is inconsistent with this file.
PAS$K_RESNOTALL	124	RESET is not allowed for an internal file that has not been opened.
PAS$K_REWNOTALL	123	REWRITE is not allowed for a file opened for sharing.
PAS$K_SUCCESS	0	Last file operation completed successfully.
PAS$K_TEXREQSEQ	13	Text files must have sequential organization and sequential access.

(continued on next page)

Table C–1 (Cont.): STATUS and STATUSV Return Values

Name	Value	Meaning
PAS$K_TRUNOTALL	125	TRUNCATE is not allowed for a file opened for sharing.
PAS$K_UPDNOTALL	126	UPDATE is not allowed for a file that has sequential organization.
PAS$K_WRIINVENU	54	WRITE operation attempted on an invalid enumerated value

Appendix D

Entry Points to VAX Pascal Utilities

This appendix describes the entry points to utilities in the VAX Run-Time Library that can be called as external routines by a VAX Pascal program. These utilities allow you to access VAX Pascal extensions that are not directly provided by the language.

D.1 PAS$FAB(f)

The PAS$FAB function returns a pointer to the RMS file access block (FAB) of file f. After this function has been called, the FAB can be used to get information about the file and to access RMS facilities not explicitly available in the VAX Pascal language.

The component type of file f can be any type; the file must be open.

PAS$FAB is an external function that must be explicitly declared by a declaration such as the following:

```
TYPE
   Unsafe_File = [UNSAFE] FILE OF CHAR;
   Ptr_to_FAB  = ^FAB$TYPE;

FUNCTION PAS$FAB
   (VAR F : Unsafe_File) : Ptr_to_FAB;
   EXTERN;
```

This declaration allows a file of any type to be used as an actual parameter to PAS$FAB. The type FAB$TYPE is defined in the VAX Pascal environment file STARLET.PEN, which your program or module can inherit.

You should take care that your use of the RMS FAB does not interfere with the normal operations of the Run-Time Library. Future changes to the Run-Time Library may change the way in which the FAB is used, which may in turn require you to change your program.

For More Information:

For information on the Run-Time Library, see the *VMS Run-Time Library Routines Volume*.

D.2 PAS$RAB(f)

The PAS$RAB function returns a pointer to the RMS record access block (RAB) of file f. After this function has been called, the RAB can be used to get information about the file and to access RMS facilities not explicitly available in the VAX Pascal language.

The component type of file f can be any type; the file must be open.

PAS$RAB is an external function that must be explicitly declared by a declaration such as the following:

```
TYPE
   Unsafe_File = [UNSAFE] FILE OF CHAR;
   Ptr_to_RAB  = ^RAB$TYPE;

FUNCTION PAS$RAB
   (VAR F : Unsafe_File) : Ptr_to_RAB;
   EXTERN;
```

This declaration allows a file of any type to be used as an actual parameter to PAS$RAB. The type RAB$TYPE is defined in the VAX Pascal environment file STARLET.PEN, which your program or module can inherit.

You should take care that your use of the RMS RAB does not interfere with the normal operations of the Run-Time Library. Future changes to the Run-Time Library may change the way in which the RAB is used, which may in turn require you to change your program.

For More Information:

For information on the Run-Time Library, see the *VMS Run-Time Library Routines Volume*.

D.3 PAS$MARK2(s)

The PAS$MARK2 function returns a pointer to a heap-allocated object of the size specified by s. If this pointer value is then passed to the PAS$RELEASE2 function, all objects allocated with NEW or PAS$MARK2 since the object was allocated are deallocated. PAS$MARK2 and PAS$RELEASE2 are provided only for compatibility with some other implementations of VAX Pascal. Their use is not recommended in a modular programming environment.

While a mark is in effect, any DISPOSE operation will not actually delete the storage, but merely mark the storage for deletion. To free the memory, you must use PAS$RELEASE2.

PAS$MARK2 is an external function that must be explicitly declared. Because the parameter to PAS$MARK2 is the size of the object (unlike the parameter to the predeclared procedure NEW), the best method for using this function is to declare a separate function name for each object you wish to mark. The following example shows how PAS$MARK2 could be declared and used as a function named Mark_Integer to allocate and mark an integer variable:

```
TYPE
   Ptr_to_Integer = ^Integer;

VAR
   Marked_Integer: Ptr_to_Integer;

[EXTERNAL(PAS$MARK2)] FUNCTION Mark_Integer
   (%IMMED S : Integer := SIZE(Integer))
   : Ptr_to_Integer;
   EXTERN;
   .
   .
   .
Marked_Integer := Mark_Integer;
```

The parameter to PAS$MARK2 can be 0, in which case the function value is only a pointer to a marker, and cannot be used to store data.

D.4 PAS$RELEASE2(p)

The PAS$RELEASE2 function deallocates all storage allocated by NEW or PAS$MARK2 since the call to PAS$MARK2 that allocated the parameter p.

PAS$MARK2 and PAS$RELEASE2 are provided only for compatibility with some other implementations of VAX Pascal. Their use is not recommended in a modular programming environment. PAS$RELEASE2 disables AST delivery during its execution, and thus should not be used in a real-time environment.

PAS$RELEASE2 is an external function that must be explicitly declared. An example of its declaration and use is as follows:

```
TYPE
   Ptr_to_Integer = ^Integer;

VAR
   Marked_Integer : Ptr_to_Integer;
```

```
[EXTERNAL(PAS$RELEASE2)] PROCEDURE Release
   (P :[UNSAFE] Ptr_to_Integer);
   EXTERN;
   .
   .
   .
Release (Marked_Integer);
```

In this example, Marked_Integer is assumed to contain the pointer value returned by a previous call to PAS$MARK2.

For More Information:

For information on PAS$MARK2, see Section D.3.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

X

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	———	Local Digital subsidiary or approved distributor
Internal[1]	———	USASSB Order Processing - WMO/E15 *or* U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

VAX Pascal Reference Supplement for VMS Systems
AA–PASYA–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

What I like best about this manual is _____

_____

What I like least about this manual is _____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

Do Not Tear - Fold Here and Tape

digital™

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

Do Not Tear - Fold Here

Cut Along Dotted Line

Reader's Comments

VAX Pascal Reference Supplement for VMS Systems
AA–PASYA–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

What I like best about this manual is _____

_____

What I like least about this manual is _____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

Do Not Tear - Fold Here and Tape

digital™

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

Do Not Tear - Fold Here

Cut Along Dotted Line

digital